

## Reading Sample

In this chapter, you'll learn about the technical basics of CDS data models. Walk through syntax elements such as key fields, cast operations, simple types, projection fields, and more. Then, familiarize yourself with associations and annotations.

-  "Fundamentals of CDS Data Modeling"
-  Contents
-  Index
-  The Authors

Colle, Dentzer, Hrastrnik

### Core Data Services for ABAP

754 pages | 10/2023 | \$89.95 | ISBN 978-1-4932-2376-3

 [www.sap-press.com/5642](http://www.sap-press.com/5642)

## Chapter 2

# Fundamentals of CDS Data Modeling

*This chapter explains the basic syntax and concepts of core data services (CDS) for defining data models. It focuses on the most important design-time artifact—the CDS view—and describes its components.*

CDS allows you to define and implement data models. The implementation of these data models is based on the CDS *data definition language* (DDL), which is closely related to the *structured query language* (SQL). It supports defining several specialized *CDS entity types* and related metadata models.

Important terms

In this chapter, [Section 2.1](#) provides an overview of these models and their purposes. The remainder of the chapter deals with CDS view models, which, from a developer's perspective, are considered the most important CDS entities. Specifically, you'll learn how to implement your *data selections* using CDS syntax. You'll find an overview of the fundamental syntax elements for CDS view definitions in [Section 2.2](#). Afterward, we'll introduce you to the following aspects in more detail:

Structure of this chapter

- Key fields ([Section 2.3](#))
- Cast operations ([Section 2.4](#))
- Typed literals ([Section 2.5](#))
- Simple types ([Section 2.6](#))
- Case statements ([Section 2.7](#))
- Session variables ([Section 2.8](#))
- Client handling ([Section 2.9](#))
- Select distinct statements ([Section 2.10](#))
- Union views ([Section 2.11](#))
- Intersect and except statements ([Section 2.12](#))
- Joins ([Section 2.13](#))
- SQL aggregation functions ([Section 2.14](#))
- Projection fields ([Section 2.15](#))
- Parameters ([Section 2.16](#))
- Reference fields ([Section 2.17](#))
- Conversion functions for currencies and quantity units ([Section 2.18](#))

- Provider contracts (Section 2.19)
- Entity buffer definitions (Section 2.20)

**Enrichments of the data selection logic**

Apart from modeling data selections, you'll also get acquainted with CDS associations and annotations. *Associations* allow you to establish directed relationships between your data models. These associations enrich the data models with semantic information. Furthermore, they can be leveraged when defining `select` statements. Associations are described in detail in [Chapter 3](#).

*Annotations* allow you to equip your CDS models with additional metadata that is primarily expected to be interpreted by the consumers of the CDS models. For example, CDS annotations are evaluated by the Service Adaptation Definition Language (SADL) infrastructure when defining an OData service. You'll learn how annotations are defined and applied in [Chapter 4](#).

## 2.1 Overview of CDS Models

CDS models comprise the definitions of various CDS entity types such as CDS views and CDS abstract entities. These entity types will cover different use cases. Typically, the CDS entity type can't be changed after the corresponding CDS model is activated. From a technical perspective, all CDS entity types are assigned to the object type DDLs, which we introduced in [Chapter 1, Section 1.2.2](#). However, the range of CDS models also comprises further objects such as CDS metadata extensions and simple types, which have deviating object types of DDLX and DRTY, respectively.

**CDS model definitions**

[Table 2.1](#) shows the fundamental definitions of the available CDS models along with their envisioned usages.

| CDS Model       | Definition                                            | Application                                 |
|-----------------|-------------------------------------------------------|---------------------------------------------|
| View            | define view ... as select from ...                    | Data selection                              |
| View entity     | define view entity ... as select from ...             | Data selection; successor of view CDS model |
| Projection view | define view entity ... as projection on ...           | Data selection                              |
| Transient view  | define transient view entity ... as projection on ... | Analytical query                            |

**Table 2.1** CDS Models

| CDS Model                | Definition                       | Application                                                                            |
|--------------------------|----------------------------------|----------------------------------------------------------------------------------------|
| Extend view              | extend view ... with ...         | Extension of a CDS view                                                                |
| Extend view entity       | extend view entity ... with ...  | Extension of a CDS view entity                                                         |
| Table function           | define table function ...        | Data selection with SAP HANA–native functions                                          |
| Custom entity            | define custom entity ...         | ABAP-based data selection in SADL-based OData services                                 |
| Extend custom entity     | extend custom entity..with..     | Extension of a CDS custom entity                                                       |
| Abstract entity          | define abstract entity ...       | Modeling of structures                                                                 |
| Extend abstract entity   | extend abstract entity..with..   | Extension of an abstract entity                                                        |
| Hierarchy                | define hierarchy entity ...      | Modeling of hierarchies                                                                |
| Metadata extension       | annotate view ... with ...       | Annotation decorator                                                                   |
| Simple type              | define type ...                  | Typing of fields and parameters; successor of ABAP Data Dictionary data element in CDS |
| Entity buffer definition | define view entity buffer on ... | Buffering records of CDS view entities                                                 |

**Table 2.1** CDS Models (Cont.)

As shown, there are different types of CDS view models:

**Views**

- **Views (aka V1 views)**  
Represent the original CDS view models.
- **View entities (aka V2 views)**  
Represent the successor of V1 views. Compared to V1 views, V2 views don't generate an additional ABAP Data Dictionary view upon their activation. This reduces the risk of technical inconsistencies and improves the overall activation performance. Furthermore, V2 views enforce more

homogenous modeling and apply stricter syntax checks. For example, V2 views foster uniform client handling, which avoids some of the issues that might occur when defining and using V1 views. Because primarily view entities are expected to benefit from further improvements of the CDS infrastructure, we recommend always defining V2 views (probably as projection or transient views) instead of V1 views.



### Migration of V1 Views to V2 Views

You can migrate your existing V1 views to V2 views. This migration can be achieved by manually changing the statement `define view ...` to `define view entity ...`. Alternatively, you can run report RUTDDLSV2MIGRATION or use the corresponding functionality **Migrate to CDS View Entity** of the context menu in the **Project Explorer** view of the ADT environment.

However, before migrating a V1 view, all the usages of its generated SQL view need to be removed. Additionally, the definitions of the V1 views and their extensions need to be aligned with the stricter syntax checks and behavior of V2 views prior to their migration. The migration tools just mentioned perform some of the necessary adjustments automatically and provide you with information about potential changes. Therefore, we suggest using the tools for migrating V1 views.

#### ■ Projection views

Represent a specialization of the view entities. Their main purpose is the definition of interfaces on their underlying CDS models with a modeled mapping of the corresponding functionality. As a result, projection views restrict the overall functionality of view entities to mere projection features. Projection views will be explained later in [Chapter 6, Section 6.1](#).

#### ■ Transient views

Transient views (sometimes also referred to as V3 views) define CDS view entities without a direct representation on the SAP HANA database system. They act as mere declarative view models whose runtime behavior is governed and implemented by infrastructure components such as the analytical engine. This implies that you can neither use transient views as data sources for other CDS views nor can you select from transient views in your ABAP code. You'll learn about the use of transient views in the context of implementing analytical queries in [Chapter 10, Section 10.3](#).

For the sake of brevity, we'll refer to *CDS view entities* when talking about CDS views or simply *views* unless explicitly mentioned otherwise.

*Extend views* and *extend view entities* allow you to enhance a CDS view or a CDS view entity, respectively, with additional fields and associations; that is, they allow you to define extensions of CDS data models. You'll learn more about such extensions in [Chapter 15](#).

View extensions

Whereas the selection logic of CDS views is implemented in the definitions of the CDS models themselves by applying a declarative CDS syntax, the implementation of *table functions* leverages native SAP HANA SQLScript syntax. The usage of SAP HANA SQLScript enables a higher degree of freedom in the way the selection logic is implemented. In addition, it provides you with the option to use functions that aren't yet supported by the CDS syntax. However, there are also several drawbacks to using table functions. You'll find further information about CDS table functions in [Chapter 7](#).

Table functions

Similar to table functions, *custom entities* only capture the signature of a data model within the CDS model definition itself. The actual implementation of custom entities occurs in ABAP code. As such, the logic of custom entities can't be executed on the database level. Instead, custom entities can be used to define OData entity sets that are processed by the SSDL infrastructure. For more information, refer to <http://s-prs.co/v529401>.

Custom entities

### ABAP Logic Can't Be Used in Logic Accessed from SAP HANA

You can't incorporate ABAP logic into the logic being executed on SAP HANA. ABAP logic can only be processed before or after processing the logic on SAP HANA.



*Abstract entities* are used to define signatures without implementation. In the context of the *ABAP RESTful application programming model*, you can use them for defining data structures for parameters and results of actions and functions, as well as for event payloads. Furthermore, abstract entities may be used for defining proxies of external service models entities. For more information, refer to the SAP documentation at <http://s-prs.co/v529421>.

Abstract entities

*Hierarchy entities* allow you to gain access to the functionality of SAP HANA hierarchies; that is, they allow you to leverage SAP HANA hierarchy features in your ABAP code. You'll find an explanation of the hierarchy modeling and its application in [Chapter 12, Section 12.3](#).

Hierarchies

Whereas all the aforementioned CDS models represent dedicated CDS entity types, *metadata extensions* represent an extension option of the CDS entity type instead. From a technical perspective, they are assigned the object type DDLX. The purpose of metadata extensions is to enrich and redefine annotations of CDS entities. The concept of metadata extensions will be explained in more detail in [Chapter 4, Section 4.4](#).

Metadata extensions

**CDS simple types** *CDS simple types* define reusable scalar type definitions. From the CDS perspective, simple types can be interpreted as successors of the ABAP Data Dictionary data elements. We'll demonstrate possible definitions and usages of simple types in [Section 2.6](#).

**Entity buffer definitions** For CDS view models with simple logic, a buffer can be defined on the application server. In this context, *entity buffer definitions* capture the required administrative data similar to table buffers that are managed by the ABAP Dictionary. Direct data selections from buffered CDS views via the ABAP SQL interface are automatically transferred to the buffer by the ABAP runtime environment, if the request can be processed on the buffer. This can help reduce the load of the SAP HANA database and speed up the provisioning of the requested data. You'll find further information about entity buffer definitions in [Section 2.20](#).

## 2.2 Overview of CDS View Syntax

**Example: CDS view definition** CDS views define select statements, which are augmented with additional metadata information. [Listing 2.1](#) provides an example of such a definition.

```
/*comment*/
//other comment
//annotations
@AccessControl.authorizationCheck: #MANDATORY
@endUserText.label: 'View Definition'
//view definition
define view entity Z_ViewDefinition
  //parameter definition
  with parameters
    P_SalesOrderType : auart
  //data source of selection with alias name
  as select from ZI_SalesOrderItem as ITEM
  //join
  left outer to exact one join ZI_SalesOrder as SO
    on SO.SalesOrder = ITEM.SalesOrder
  //association definition
  association [0..1] to ZI_Product as _Product on
    $projection.RenamedProduct = _Product.Product
{
  //projected field as key
  key ITEM.SalesOrder,
  //projected field used in association definition
  key ITEM.Product as RenamedProduct,
```

```
//constant
abap.char 'A' as Constant,
//calculated field
concat( ITEM.SalesOrder, ITEM.Product ) as CalculatedField,
//aggregate
count(*) as NumberOfAggregatedItems,
//projected association
ITEM._SalesOrder,
//association exposure
_Product
}
//filter conditions based on join partner and parameter
where
  SO.SalesOrderType = $parameters.P_SalesOrderType
//aggregation level
group by
  ITEM.SalesOrder,
  ITEM.Product
```

### Listing 2.1 CDS View Definition

Let's now take a closer look at this definition:

Example analysis

- **define statement**  
The define statement specifies the technical type of the CDS model. In the given example, a CDS view entity model named Z\_ViewDefinition is defined (define view entity).
- **Data sources**  
CDS view Z\_ViewDefinition uses the CDS view for sales order items (ZI\_SalesOrderItem) as its primary data source (select from). It combines this primary data source via a left outer join statement with a secondary data source, which provides the header data of sales orders ZI\_SalesOrder. Both data sources are locally renamed by alias names ITEM and SO, respectively. You'll learn more about join types in [Section 2.13](#).
- **Projected elements**  
CDS view Z\_ViewDefinition projects various elements from its primary data source, for instance, field SalesOrder, which identifies the sales order, and field Product, which establishes a relationship between the sales order item and its assigned product. The latter field is renamed by aliasing it in RenamedProduct.
- **Aggregation**  
Due to the applied aggregation logic of the data records (group by) by

fields `SalesOrder` and `Product`, both of these fields determine the key of CDS view `Z_ViewDefinition`.

#### ■ Calculated fields

Besides the projected fields, CDS view `Z_ViewDefinition` also defines the field `Constant` locally, which has the value `A` for all records. Technically, this field is defined as a typed literal (Section 2.5), which is based on the elementary ABAP Data Dictionary type `char` of length 1. Furthermore, it contains field `CalculatedField`, which concatenates the values of fields `SalesOrder` and `Product` by applying function `concat`.

Field `NumberOfAggregatedItems` represents an aggregate. Its value is determined by counting (`count(*)`) the records of sales order items, which are grouped in a single record of the selection result.

#### ■ where condition

The data selection is filtered by a `where` clause, which compares the sales order type (`SalesOrderType`) with the value of CDS parameter `P_SalesOrderType`.

#### ■ Association

The CDS view model also defines association `_Product`, which establishes a named relationship to the corresponding record of the CDS view of product header `ZI_Product`. The association is exposed by incorporating it in the projection list of the view. As a result, the association can be used by consumers of the CDS view too.

#### ■ Comments

Within the CDS view model, *comments* refer to single-line comments prefixed with double slashes (`//`) or incorporated into the pattern `/*...*/`, which allows you to define multiline comments.

#### ■ Annotations

Annotation `@AccessControl.authorizationCheck: #MANDATORY` indicates that the selection result will be subject to an access control. You'll learn more about how to enable such authorization checks in Chapter 5. Annotation `@EndUserText.label` specifies a language-dependent text for the view that will correspond to its description.

#### Detailed information

As you can see from the presented example, the CDS syntax consists of various different elements. To master the definition of CDS models yourself, we'll provide you with more detailed information about these syntax elements in the subsequent sections and chapters of this book.

#### Technical aspects

The definition of a CDS view model is captured as source code in the transportable object of type `DDL`. The CDS view isn't transported; instead, it's

generated locally when activating the `DDL` object. In the ABAP repository, it has object type `STOB` (CDS view). For CDS view entities, the name of the CDS view (converted to uppercase letters) has to match the name of its defining `DDL` object.

#### Consider the Namespace of CDS Models

The CDS view name shares the same namespace with other ABAP Data Dictionary artifacts (e.g., tables and views). This restriction has to be considered when choosing names for CDS views and `DDL` objects.

In addition, to avoid name clashes with SAP-delivered artifacts, you should always use your dedicated customer or partner namespace when defining your own CDS models.

## 2.3 Key Fields

CDS view fields are defined as *key fields* by adding the preceding syntax element `key` to the fields. In general, the key of a CDS model can be composed of multiple key fields. These key fields must be placed before the non-key fields in the projection list of the CDS view. The key must be defined in such a way that it uniquely identifies a single record in the results list of a data selection. It's recommended to keep the key as short as possible; that is, fields should only be included in the key definition if they are required for achieving an unambiguous identification of a single record in the results list of a data selection.

#### Client Field

CDS view entities apply an automated filtering based on the client (Section 2.9). Therefore, the client field isn't contained in the list of key fields, even if the underlying database records depend on the client.

The sample CDS view for sales order item `ZI_SalesOrderItem` in Listing 2.2 has a key that consists of the two fields `SalesOrder` and `SalesOrderItem`. Both fields are required key components because the identifier of the sales order item is only uniquely defined in the context of its embedding sales order document. From a technical perspective, it's also possible to mark field `Product` as another key field. However, this field isn't required for making the key of the CDS view unique, so it isn't included in the illustrated key definition.

Key definition

Example: CDS view with key fields

```

define view entity ZI_SalesOrderItem
  as select from zsalesorderitem
{
  key salesorder      as SalesOrder,
  key salesorderitem as SalesOrderItem,
  product            as Product
}

```

Listing 2.2 CDS View with Key Definition

## Using the key definition

The key definition serves as documentation of a CDS view model. In addition, it can be used for consistency checks. For example, you can check the specified cardinality of an association based on the bound key fields of the target CDS model. Key definitions may also have an impact on the CDS access controls that evaluate the key definitions when injecting authorization restrictions into `select` statements. Furthermore, it's evaluated by various implementation frameworks, which provide their services on top of the corresponding CDS view. For instance, the key definition of a CDS model can be used for automatically deriving the key definition of an *odata entity set*, which is mapped onto the CDS model. [Chapter 6](#) provides you with more information about the OData exposure of CDS models.



## Increase Transparency of CDS Models

You should always define a key for your CDS models if feasible from a technical perspective. A key definition should only be omitted if the CDS model doesn't contain suitable fields or combinations for defining a unique key.

## 2.4 Cast Operations

## Cast operations

*Cast operations* represent some of the most important fundamental SQL functions. You can use cast operations for determining the type of a calculated field and for converting the type of existing fields on the database level. The CDS syntax supports casts onto elementary ABAP types as well as onto data elements and simple types. By using data elements or simple types, you not only can change the type of the cast fields but also assign suitable annotations and properties such as label texts and conversion exits to them.

## Cast operations without type changes

If a cast doesn't change the technical type of the field but, for example, is performed to exchange its label texts, you can enrich the cast statement with addition `preserving type`. This addition informs the compiler that no

effective type change is required, and, as a result, no cast is required on the database level.

## Supported Type Conversions

Not all type conversions are supported by the CDS syntax. You'll find an overview of the supported conversions in your ABAP documentation, for example, at <http://s-prs.co/v529402>.

Some type conversions aren't supported by a cast operation and can only be achieved by applying dedicated CDS conversion functions. For example, you can only convert a floating point value to a decimal value by using function `fltp_to_dec`.

In a CDS table function implementation, the extended options of native SAP HANA conversion functions are available. Thus, engaging a CDS table function may allow you to perform conversions that aren't supported by the CDS syntax itself.

[Listing 2.3](#) shows a CDS view with fields that have different type conversions.

```

define view entity Z_ViewWithCasts as select distinct from t000
{
  t000.logsys                                as ProjectedField,
  '20170809'                                 as CharacterField,
  cast ( '20170809' as abap.dats )           as DateField,
  cast ( cast ( 'E' as abap.lang ) as sylangu preserving type )
  as LanguageField,
  1.2 as FloatingPointField,
  fltp_to_dec( 1.2 as abap.dec(4,2) ) as DecimalField
}

```

Listing 2.3 CDS View with Cast Operations

By default, a field keeps the type of its origin if it's simply projected into a CDS view. For example, field `ProjectedField` retains the type from its underlying base field `t000.logsys`.

If fields are locally defined, an implicit type assignment is applied. As a result, field `CharacterField` is defined as a `char` field of length 8 based on the literal value `20170809`. By explicitly casting the same literal value onto type `dats`, field `DateField` becomes a date field.

You can also nest cast operations. This option is illustrated by expression `cast ( cast ( 'E' as abap.lang ) as sylangu preserving type )`. The value `E` is implicitly regarded as a character value of length 1. In a first step, the type



Example:  
CDS view with type conversions

Example analysis

of corresponding field `LanguageField` is explicitly set to `lang`. In a second step, data element `sy langu` finally determines the type of the field. Because there is no change of the technical type, this conversion is accompanied by addition preserving type.

By assigning value 1.2 to field `FloatingPoint`, this field is implicitly typed by `fltp`. In addition, field `DecimalField` is constructed from value 1.2. Due to this value assignment, it's implicitly defined as a field of type `fltp` too. To convert this field into a field of type `dec`, function `fltp_to_dec` is applied.



### Use Explicit Type Assignment

You should always explicitly cast calculated fields onto the desired data type if feasible from a technical perspective. Otherwise, you may experience unexpected side effects from implicit type assignments.

If you're not sure about the type of a CDS field, you can position your cursor on the CDS view or one of its fields and press `[F2]`. This launches an information popup, as shown in [Figure 2.1](#), where you can find the previously discussed type information.

| Column             | Data Element | Data Type   | Description    |
|--------------------|--------------|-------------|----------------|
| ProjectedField     | logsys       | char(10)    | Logical system |
| CharacterField     |              | numc(8)     |                |
| DateField          |              | dats(8)     |                |
| LanguageField      | sy langu     | lang(1)     | Language Key   |
| FloatingPointField |              | fltp(16,16) |                |
| DecimalField       |              | dec(4,2)    |                |

Figure 2.1 Technical Field Information of the CDS View from [Listing 2.3](#)

## 2.5 Typed Literals

**Typed literals** *Typed literals* allow you to specify the technical ABAP type of a constant value, which you introduce in your CDS model. To make a plain literal become a typed literal, you have to enclose the actual value in single quotation marks and add the type information as a prefix. [Listing 2.4](#) shows some examples.

```
define view entity Z_ViewWithTypedLiterals
  as select distinct from t000
{
  ' Char10 ' as CharacterValue,
  cast( ' Char10 ' as abap.char(10) ) as CastCharacterValue,
  abap.char' Char10 ' as TypedCharacterValue,
  cast( abap.char' Char10 ' as Char10 preserving type )
  as CastTypedCharacterValue,
  1234.56 as FloatingPointValue,
  abap.fltp'1234.56' as TypedFloatingPointValue,
  fltp_to_dec(1234.56 as abap.dec(6,2)) as ConvertedDecimalValue,
  abap.dec'1234.56' as TypedDecimalValue,
  abap.dec'001234.5600' as TypedDecimalValue2
}
```

Listing 2.4 CDS View with Typed Literals

In this example, field `CharacterValue` is derived from literal `' Char10 '`. From the CDS perspective, this field is implicitly typed as a character field of length 8; that is, the leading two spaces are considered, whereas the last two spaces are ignored.

With an explicit cast, you can enforce the field to have a length of 10 characters. This is shown by field `CastCharacterValue`. Similarly, field `TypedCharacterValue`, which is derived from the typed literal `abap.char' Char10 '`, becomes a character field of length 10 automatically. In other words, for typed literals, the entire value entered in the quotation marks is considered to be significant.

If you define a field based on a typed literal and want to equip it with additional metadata such as label texts, you should wrap the typed literal with an explicit type-preserving cast onto a suitable data element or simple type. This is illustrated by field `CastTypedCharacterValue`.

Typed literals become especially important if the literals don't represent mere character strings. This will be demonstrated by the remaining fields that are defined in [Listing 2.4](#).

When defining a value such as `1234.56`, it's implicitly interpreted as a floating point value. Consequently, `FloatingPointValue` and `TypedFloatingPointValue` share the same technical typing. If you want to define a decimal value instead, you basically have two options: (1) perform an explicit type conversion of such a floating point value by applying function `fltp_to_dec`, as illustrated for field `ConvertedDecimalValue`; or (2) define a typed literal. Field `TypedDecimalValue` in [Listing 2.4](#) is typed accordingly. However, note that `abap.dec'1234.56'` is interpreted as a decimal value from the very

Example: CDS view with typed literals

Example analysis

Character like values

Non-character like values



beginning. In contrast, floating point value 1234.56 has to be effectively converted to a decimal value. This not only implies that there is some overhead in the processing logic of the value but also may be the reason the field values returned by a view differ. In our case, `TypedDecimalValue` exposes modeled value '1234.56', whereas `ConvertedDecimalValue` returns converted value '1234.55'.

As mentioned before, the value and its notation are specified within the typed literal matter. Typed literal `abap.dec'1234.56'` is associated with a decimal type of length 6 with 2 decimals. Typed literal `abap.dec'001234.5600'` yields from typing field `TypedDecimalValue2` as a decimal field of length 10 with 4 decimals.



### Use Typed Literals

We recommend always using typed literals instead of plain literals. They help you avoid making errors and introducing unnecessary conversions.

Influence type length

As already explained the concrete value defined in a typed literal is decisive. For example, typed literal `abap.dec'1234.56'` results in field `TypedDecimalValue` having type `dec` of length 6 with 2 decimals. In contrast, typed literal `abap.dec'001234.5600'` results in field `TypedDecimalValue2` being defined as a decimal field with length 10 and 4 decimals.

## 2.6 Simple Types

Simple type definitions

CDS simple types can be specified based on different type definitions: [Listing 2.5](#) illustrates simple type `ZBT_LanguageA`, which is based on elementary dictionary type `lang`.

```
define type ZBT_LanguageA : abap.lang;
```

**Listing 2.5** Simple Type Based on an Elementary ABAP Data Dictionary Type

[Listing 2.6](#) shows simple type `ZBT_LanguageB`, which is based on data element `spras`.

```
define type ZBT_LanguageB : spras;
```

**Listing 2.6** Simple Type Based on a Data Element

[Listing 2.7](#) demonstrates the definition of simple type `ZBT_LanguageC`, which is based on the simple type `ZBT_LanguageB` from [Listing 2.6](#).

```
@EndUserText.label: 'Language Type C'
@Semantics.language: true
define type ZBT_LanguageC : ZBT_LanguageB;
```

**Listing 2.7** Simple Type Based on Another Simple Type

In addition, [Listing 2.7](#) shows how you can enrich the mere technical definition of a simple type with annotations. In the given case, simple type `ZBT_LanguageC` is classified as a language code by its annotation `@Semantics.language: true`. In addition, it receives the label text 'Language Type C' from its annotation `@EndUserText.label`.

Enrichment with annotations

You can define multilevel hierarchies of simple types, as shown in [Listing 2.8](#). Therein, simple type `ZBT_LanguageD` is typed by simple type `ZBT_LanguageC` from [Listing 2.7](#).

Multilevel definitions of simple types

```
@EndUserText.label: 'Language Type D'
@ObjectModel.sapObjectTypeReference: 'Language'
define type ZBT_LanguageD : ZBT_LanguageC;
```

**Listing 2.8** Simple Type Based on Another Simple Type: Multilevel

In this context, the annotations of simple type `ZBT_LanguageC` are propagated to simple type `ZBT_Language` such as the technical properties of the underlying data element `spras`. These annotations are overlaid by the local annotations of simple type `ZBT_LanguageD`. You'll learn more about the resulting active annotations in [Chapter 4, Section 4.3.1](#).

You can leverage simple types for typing fields and parameters ([Section 2.16](#)) of your CDS models. In [Listing 2.9](#), CDS view field `Language1` is typed by simple type `ZBT_LanguageA`, and field `Language2` is typed by simple type `ZBT_LanguageD`. These fields take over the technical properties, including the annotations of their typing simple types.

Simple types in CDS models

```
define view entity Z_ViewWithSimpleTypes
  as select distinct from t000
{
  cast ( abap.lang'E' as ZBT_LanguageA preserving type ) as Language1,
  cast ( abap.lang'E' as ZBT_LanguageD preserving type ) as Language2
}
```

**Listing 2.9** CDS View with Fields Typed by Simple Types



### Incorporate Annotations by Simple Types into CDS Models

You can use simple types for incorporating field-specific or type-specific CDS annotations into your CDS models. In this context, simple types act as reusable metadata containers, which support you in efficiently defining consistent CDS models.

## 2.7 Case Statements

**Case statements** You can use *case statements* for defining conditional calculations in your CDS view logic. Within a single case statement, you can define multiple execution paths via *when-then* switches. Based on the *when* condition, you can apply dedicated *then* instructions, which finally determine the calculated value of a field. At the end of a case statement, you can add an *else* instruction without conditions. This branch allows you to handle cases that aren't covered by the *when* conditions. Without this fallback valuation, the calculated field may have the value *null*, if none of the explicit *when* conditions match.



### Avoid Non-Null Preserving Expressions

Unconditional *else* branches with assignments of constant values can result in non-null preserving expressions. Such expressions may prevent the SAP HANA optimizer from reordering the execution plan for the most efficient processing of the *select* statement, which in turn can result in performance issues.

**Example:** Listing 2.10 provides an example for a CDS view with three case statements.  
**CDS view with case statements**

```
define view entity Z_ViewWithCaseStatements
as select from ZI_SalesOrder
{
  key SalesOrder,
  case (SalesOrderType)
    when 'TAF' then 'X'
    when 'OAF' then 'X'
    else ''
  end as IsStandardOrder,
  cast( case (SalesOrderType)
    when 'TAF' then 'X'
    when 'OAF' then 'X'
```

```
    else ''
  end as abap.char(3) ) as IsStandardOrderAsChar3,
  case when SalesOrderType = 'TAF' then 'X'
    when SalesOrderType = 'OAF' then 'X'
    else ''
  end as IsStandardOrder2
}
```

**Listing 2.10** CDS View with Case Statements

The first case statement checks the value of field *SalesOrderType* and sets the value of field *IsStandardOrder* to the constant *X* if the sales order type has a value of *TAF* or *OAF*. In all other cases, the value is set to the initial value. In the given case, the calculation implicitly results in a field of type *char* with length 1.

You can influence the resulting type explicitly using a wrapping cast operation. This is illustrated for the second case statement, which uses the same logic as the first case statement. Due to the enclosing cast operation, field *IsStandardOrderAsChar3* receives type *char* with length 3.

The third case statement contains the same logic as the first case statement. From a principle perspective, the *when* conditions of the branches of this third case statement may also implement complex rules, for example, by applying pattern comparisons with *like* as well as by combining multiple conditions with *and* and *or* operators. In contrast, the *when* conditions in the first case statement only support simple comparisons of operands.

**Example analysis**

## 2.8 Session Variables

*CDS session variables* allow you to access information regarding the current runtime session within the CDS view logic. Similar to literal values, you can use session variables at various places within the implementation of your data selections.

Some standard usages are depicted in Listing 2.11, in which a filter is defined by *and* fields are derived from session variables.

```
define view entity Z_ViewWithSessionVariables
as select from t000
{
  $session.client as ClientField,
  $session.system_date as SystemDateField,
  $session.system_language as SystemLanguageField,
```

**Example:**  
CDS view with session variables

```

$session.user      as UserField,
$session.user_date as UserDateField,
$session.user_timezone as UserTimezoneField
}
where
  mandt = $session.client

```

Listing 2.11 CDS View with Session Variables

Table 2.2 provides you with an overview of the session variables that are supported by the CDS syntax.

| Session Variable          | Description                           | Correspondence in ABAP Code |
|---------------------------|---------------------------------------|-----------------------------|
| \$session.client          | Current client                        | sy-mandt                    |
| \$session.system_date     | System date of the application server | sy-datum                    |
| \$session.system_language | Logon language                        | sy-langu                    |
| \$session.user            | Current user                          | sy-uname                    |
| \$session.user_date       | Current user date                     | sy-datlo                    |
| \$session.user_timezone   | User time zone                        | sy-zonlo                    |

Table 2.2 Session Variables



### Session Variables versus Parameters

If you use session variables, the data selection of your CDS view is influenced by factors that a consumer of your CDS view may not be able to control explicitly. This can result in unwanted restrictions and make the selection results more difficult to interpret.

In many cases, you can replace session variables with parameters (Section 2.16), which eases the error analysis and maintenance of CDS views. However, introducing parameters may require the consumers of a CDS view to adapt. In contrast, the introduction of session variables is a local implementation detail of a view. Typically, from a technical perspective, it's compatible for ABAP consumers. Only when introducing a client dependency by using session variable \$session.client may adaptations be required.

## 2.9 Client Handling

When performing standard selections on database tables via the ABAP SQL interface, client-specific data is automatically filtered by the current client of the ABAP session. To perform cross-client data accesses, either addition using client or addition client specified must be added to the select statement. Listing 2.12 shows an example of such a selection, where the data is read from client 001.

Standard data selection in ABAP

```

SELECT *
  FROM zsalesorder
  CLIENT SPECIFIED
 INTO TABLE @DATA(lt_salesorder)
 WHERE client = '001'.

```

Listing 2.12 Cross-Client Data Selection in ABAP

In CDS view entities, the standard client handling is enforced by the ABAP Data Dictionary infrastructure. This is achieved by automatically augmenting the technical view definition on SAP HANA with an additional where condition, if this is required.

Client handling in view entities

### Consider Restrictions Imposed by Access Controls

Cross-client data accesses via CDS views aren't allowed if the CDS views are protected by access controls (see Chapter 5).



Listing 2.13 and Listing 2.14 illustrate an example. The CDS view entity ZI\_Product in Listing 2.13 selects from a client-dependent table ZPRODUCT.

Example: Client-dependent CDS view entity

```

define root view entity ZI_Product
  as select from zproduct
{
  key product          as Product,
  product_type        as ProductType,
  creation_date_time as CreationDateTime
}

```

Listing 2.13 CDS View Selecting from a Client-Specific Data Source

Even though it doesn't explicitly handle the client dependency, its generated database view maps the client field CLIENT onto context variable CDS\_CLIENT, as depicted in Listing 2.14. This context variable is associated with the session variable \$session.client, thus the selection result is restricted accordingly.

```
CREATE OR REPLACE VIEW "ZI_PRODUCT" AS SELECT
  "ZPRODUCT"."CLIENT" AS "MANDT",
  "ZPRODUCT"."PRODUCT" AS "PRODUCT",
  "ZPRODUCT"."PRODUCT_TYPE" AS "PRODUCTTYPE",
  "ZPRODUCT"."CREATION_DATE_TIME" AS "CREATIONDATETIME"
FROM "ZPRODUCT" "ZPRODUCT"
WHERE "ZPRODUCT"."CLIENT" = SESSION_CONTEXT(
  'CDS_CLIENT'
)
```

**Listing 2.14** Create Statement of the Database View Generated from [Listing 2.13](#)

**Client field** Because the client is fixed, the client field shouldn't be included in the projection lists of the CDS views. However, CDS views, whose records only differ in their client fields, are an exception.

The client field should also be contained in the signatures of CDS table functions (see [Chapter 7](#)) that access client-specific data records. For more information about this topic, see the ABAP documentation at <http://s-prs.co/v529403>.

## 2.10 Select Distinct Statements

**Condense selection result** By applying the *select distinct statement* (`select distinct`), you can remove duplicate records from the selection result list and thus condense it. The comparison considers the values of all requested fields. Let's take a look at the sample CDS views in [Listing 2.15](#) and [Listing 2.16](#).

**Example: CDS view without select distinct** CDS view `Z_ViewWithoutSelectDistinct` in [Listing 2.15](#) uses table T000, which contains one record per client. Its implemented logic results in multiple selected records (one per client), which share the value A for field `Field1`. Correspondingly, no key can be specified for this CDS view.

```
define view entity Z_ViewWithoutSelectDistinct
  as select from t000
{
  abap.char'A' as Field1
}
```

**Listing 2.15** CDS View without Select Distinct Statement

In contrast, CDS view `Z_ViewWithSelectDistinct` in [Listing 2.16](#) combines the previous selection logic with a `distinct` statement. Independent from the number of setup clients, that is, the number of records provided by

table T000, CDS view `Z_ViewWithSelectDistinct` returns a single record (there is at least one record in table T000). Therefore, `Field1` can be marked as a key field.

```
define view entity Z_ViewWithSelectDistinct
  as select distinct from t000
{
  key abap.char'A' as Field1
}
```

**Listing 2.16** CDS View with Select Distinct Statement

## 2.11 Union Views

*Union views* combine and unify data records of different data sources. The outcome of such a union is a results list that comprises all data records of the unified data sources and that is harmonized from the perspective of its consumers, providing them with uniform fields and associations. In [Section 2.11.1](#), you get an overview about the fundamental modeling of union CDS views. In [Section 2.11.2](#), we explain the differences between the plain union and the union all operators.

### 2.11.1 Union Definitions

You define union views by combining multiple `select` statements with the `union` statement. Each individual selection branch must define the same fields and associations in the same order. In addition, the corresponding items of the selection lists must have the same definitions. This implies that the names of the elements within the individual branches must be the same. Furthermore, the underlying association definitions must specify the same on conditions, cardinalities, and target entities. The key definitions must be the same across all branches too. However, the types of the individual fields of the union view may deviate for each branch, if they are convertible. In such a case, the effective type of the field of the union view is derived from the corresponding field of the first `select` statement.

Similarly, element annotations are determined by the first selection branch. Therefore, only elements of the first selection branch may be annotated. Element annotations must not be propagated to a union view. This requires annotating the union view with `@Metadata.ignorePropagatedAnnotations: true`. [Chapter 4, Section 4.3](#) explains the propagation logic in more detail.

Modeling  
union views

**Example:** Let's look at the following example, which comprises four CDS view definitions. CDS view `Z_ViewAsDataSourceA` from [Listing 2.17](#) acts as a data source for union CDS view `Z_UnionView` from [Listing 2.20](#), later in this chapter.

```
define view entity Z_ViewAsDataSourceA
  as select distinct from t000
  association [0..1] to Z_ViewAsDataSourceC as _ViewC
  on $projection.FieldA3 = _ViewC.FieldC1
{
  key cast( 'A' as abap.char(1) ) as FieldA1,
    cast( 'B' as abap.char(1) ) as FieldA2,
    cast( 'C' as abap.char(2) ) as FieldA3,
    _ViewC
}
```

**Listing 2.17** CDS View `Z_ViewAsDataSourceA`

Likewise, CDS view `Z_ViewAsDataSourceB` from [Listing 2.18](#) acts as a data source for union CDS view `Z_UnionView` from [Listing 2.20](#).

```
define view entity Z_ViewAsDataSourceB
  as select distinct from t000
{
  key cast( 'B_X' as abap.char(3) ) as FieldB1,
    cast( 'A' as abap.char(1) ) as FieldB2
}
```

**Listing 2.18** CDS View `Z_ViewAsDataSourceB`

**Example:** CDS view `Z_ViewAsDataSourceC` from [Listing 2.19](#) is used as the association target of view `Z_ViewAsDataSourceA` from [Listing 2.17](#) and union CDS view `Z_UnionView` from [Listing 2.20](#).

```
define view entity Z_ViewAsDataSourceC
  as select distinct from t000
{
  key cast( 'C' as abap.char(2) ) as FieldC1,
    cast( 'C2' as abap.char(2) ) as FieldC2
}
```

**Listing 2.19** CDS View `Z_ViewAsDataSourceC`

**Analysis of the sample views** Each of the mentioned CDS views defines a single data record. For example, CDS view `Z_ViewAsDataSourceA` selects a single data record from table `T000`, which is always populated, by applying the `distinct` statement. It returns

the constant values A, B, and C for its `FieldA1`, `FieldA2`, and `FieldA3`, respectively. Similarly, the other two CDS views, `Z_ViewAsDataSourceB` and `Z_ViewAsDataSourceC`, return single records with constant field values. [Table 2.3](#) shows an overview of the data provided by these CDS views.

| CDS View                         | FieldA1/B1/C1 | FieldA2/B2/C2 | FieldA3 |
|----------------------------------|---------------|---------------|---------|
| <code>Z_ViewAsDataSourceA</code> | A             | B             | C       |
| <code>Z_ViewAsDataSourceB</code> | B_X           | A             | -       |
| <code>Z_ViewAsDataSourceC</code> | C             | C2            | -       |

**Table 2.3** Data Records of the CDS Views from [Listing 2.17](#) to [Listing 2.19](#)

Union CDS view `Z_UnionView` in [Listing 2.20](#) combines CDS views `Z_ViewAsDataSourceA` and `Z_ViewAsDataSourceB` as its data sources. In addition, it exposes association `_ViewC` to CDS view `Z_ViewAsDataSourceC`.

**Example:** Union CDS view

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionView
  as select from Z_ViewAsDataSourceA
  association [0..1] to Z_ViewAsDataSourceC as _ViewC on
  $projection.UnionField1 = _ViewC.FieldC1
{
  @EndUserText.label: 'Label of UnionField1'
  key FieldA1 as UnionField1,
  key FieldA2 as UnionField2,
  key FieldA3 as UnionField3,
  _ViewC
}
union select from Z_ViewAsDataSourceB
  association [0..1] to Z_ViewAsDataSourceC as _ViewC on
  $projection.UnionField1 = _ViewC.FieldC1
{
  key FieldB2 as UnionField1,
  key FieldB1 as UnionField2,
  key '' as UnionField3,
  _ViewC
}
```

**Listing 2.20** Union CDS View `Z_UnionView`

To harmonize the element names, the fields contained in the individual selection branches of union CDS view `Z_UnionView` are mapped to field names `UnionField1`, `UnionField2`, and `UnionField3` using alias function `as`. All

**Analysis of sample union CDS view**

fields contained in the field list of the union CDS view have type `char` with length 1 or 2. This definition is derived from the definition of underlying base fields `FieldA1`, `FieldA2`, and `FieldA3` of the first selection statement of the union CDS view. [Table 2.4](#) shows the resulting records of the union CDS view.

| UnionField1 | UnionField2 | UnionField3   |
|-------------|-------------|---------------|
| A           | B           | C             |
| A           | B           | Initial value |

**Table 2.4** Records of the Union CDS View `Z_UnionView`

Note that `FieldB1` of CDS view `Z_ViewAsDataSourceB`, which has a maximum length of 3, is implicitly shortened when it's mapped onto `UnionField1`. Therefore, the corresponding data record of the union CDS view contains value `B` instead of original value `B_X` (compare [Table 2.3](#) with [Table 2.4](#)). If the types of the mapped fields aren't automatically mutually convertible, you must explicitly define a type conversion. This can typically be achieved by applying a `cast` operation ([Section 2.4](#)) for the fields in question.

In general, each selection branch of a union view must define the same number of fields, so `UnionField3` must also be inserted into the second selection branch. Because there is no related information available in underlying base CDS view `Z_ViewAsDataSourceB`, this field is filled with the initial value, which corresponds to its type.

In the first selection branch, `UnionField1` is annotated with `@EndUserText.label...` Such annotations are valid for all other branches too; that is, the annotations of the first branch determine the corresponding annotations of the union CDS view as a whole.

#### Associations in union CDS views

In union CDS view `Z_UnionView` from [Listing 2.20](#), association `_ViewC` to view `Z_ViewAsDataSourceC` defined in its base view `Z_ViewAsDataSourceA` can't simply be projected into the first selection branch because the definition of association `_ViewC` in the second selection branch of union view `Z_UnionView` has a different `ON` condition than in CDS view `Z_ViewAsDataSourceA`. Due to the requirement that all selection branches must share the same association definitions in union views, this association must be redefined in the first selection branch too.

#### Two-layer union CDS views

To avoid redundancies in association definitions, you could split the definition of your union CDS views into two CDS views that are built one on top of the other. [Listing 2.21](#) and [Listing 2.22](#) show an example of such a two-layer construction.

CDS view `Z_UnionViewWithoutAssociations` from [Listing 2.21](#) implements union logic.

**Example:**  
Union CDS view  
without association

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithoutAssociations
  as select from Z_ViewAsDataSourceA
  {
    @EndUserText.label: 'Label of UnionField1'
    key FieldA1 as UnionField1,
    key FieldA2 as UnionField2,
    key FieldA3 as UnionField3
  }

union select from Z_ViewAsDataSourceB
  {
    key FieldB1 as UnionField1,
    key FieldB2 as UnionField2,
    key '' as UnionField3
  }
```

**Listing 2.21** Union CDS View without Associations

CDS view `Z_UnionViewWithAssociations` from [Listing 2.22](#), which is based on this intermediate view, enriches union logic with association `_ViewC` without introducing redundancies in the definition.

**Example:**  
Dependent  
CDS view with  
association

```
define view entity Z_UnionViewWithAssociations
  as select from Z_UnionViewWithoutAssociations
  association [0..1] to Z_ViewAsDataSourceC as _ViewC
  on $projection.UnionField1 = _ViewC.FieldC1
  {
    key UnionField1,
    key UnionField2,
    key UnionField3,
    _ViewC
  }
```

**Listing 2.22** CDS View with Association Based on Union CDS View

#### Association Definitions in Union CDS Views



If you need to define a larger number of associations within your union CDS view, it may be beneficial to relocate the association definitions to a separate superordinated CDS view. This CDS view can transfer the field list of the union CDS view via the regular projection mechanism and define the required associations once locally.

### 2.11.2 Union and Union All Logic

**Union logic** When applying union logic, duplicate records, which originate from the different merged data sources, are automatically removed from the results list. For example, by applying union logic on top of the same data source `Z_ViewAsDataSourceA` from [Listing 2.17](#), the implementation of CDS view `Z_UnionViewWithoutDuplicat` from [Listing 2.23](#) leads to a single resulting data record.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithoutDuplicat
  as select from Z_ViewAsDataSourceA
  {
    key FieldA1
  }
union select from Z_ViewAsDataSourceA
  {
    key FieldA1
  }
```

**Listing 2.23** Union CDS View without Duplicate Records (Union Logic)

**Union All logic** If this isn't desired, you should use union all logic, which retains the data records of the data sources involved. In contrast to the selection in [Listing 2.23](#), by applying union all logic, the selection result of CDS view `Z_UnionViewWithDuplicat` from [Listing 2.24](#) contains two identical data records.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewWithDuplicat
  as select from Z_ViewAsDataSourceA
  {
    FieldA1
  }
union all select from Z_ViewAsDataSourceA
  {
    FieldA1
  }
```

**Listing 2.24** Union CDS View with Duplicate Records (Union All Logic)



#### Performance Implications

Due to the missing comparison and filtering of data records, union all logic typically outperforms union logic. If applicable, you should therefore choose the union all logic.

Merging selection results from various data sources requires a critical check of the key definitions of the union CDS views. If CDS views that have their own key definitions are merged, the key of the union CDS view isn't necessarily the same as the superset of all key fields of the merged CDS views. In union CDS view `Z_UnionView` from [Listing 2.20](#), not only are `UnionField1` and `UnionField2` part of the key but also `UnionField3`. In the given example, `UnionField3` is required for a unique differentiation between the two data records of the selection result (refer to [Table 2.4](#)). If the second selection branch of the union view would set this field to value C instead of setting it to the initial value, both selection branches would return the same data record. In this case, you wouldn't be able to designate a unique key for union CDS view `Z_UnionView` as a whole.

In general, a unique key can't be specified for a CDS view with duplicate records. Therefore, the key definition should always be omitted in such a CDS view (refer to [Listing 2.24](#)).

## 2.12 Intersect and Except Statements

Besides merging records of data sources, the CDS syntax also supports you in defining intersections of and exceptions from data sources, which will be illustrated in this section.

[Listing 2.25](#) shows CDS view `Z_UnionViewAsDataSourceA`, which will act as a data source. It returns two records as given in [Table 2.5](#).

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewAsDataSourceA
  as select distinct from t000
  {
    key 'A' as Field1
  }
union all select distinct from t100 {
  key 'B' as Field1
}
```

**Listing 2.25** CDS View `Z_UnionViewAsDataSourceA`

| Z_UnionView-AsDataSourceA | Z_UnionView-AsDataSourceB | Z_ViewWith-Intersect | Z_ViewWith-Except |
|---------------------------|---------------------------|----------------------|-------------------|
| A                         | A                         | A                    | B                 |
| B                         | C                         | -                    | -                 |

**Table 2.5** Values of Field1 of the CDS Views Records from [Listing 2.25](#) to [Listing 2.28](#)

Key definitions

Example:  
CDS views serving  
as data sources

Similarly, [Listing 2.26](#) shows CDS view `Z_UnionViewAsDataSourceB`, which will act as another data source. It also returns two records, as given in [Table 2.5](#).

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_UnionViewAsDataSourceB
  as select distinct from t000
{
  key 'A' as Field1
}
union all select distinct from t100 {
  key 'C' as Field1
}
```

**Listing 2.26** CDS View `Z_UnionViewAsDataSourceB`

**Example: Intersection** If you want to determine those records that two data sources have in common, you can use the CDS syntax element `intersect`. CDS view `Z_ViewWithIntersect` from [Listing 2.27](#) defines such an intersection of CDS view `Z_UnionViewAsDataSourceA` from [Listing 2.26](#) and CDS view `Z_UnionViewAsDataSourceB` from [Listing 2.25](#). Both these data sources contain a record with the value A for `Field1`, which represents the result of the intersection (refer to [Table 2.5](#)).

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewWithIntersect
  as select from Z_UnionViewAsDataSourceA
{
  key Field1
}
intersect select from Z_UnionViewAsDataSourceB
{
  key Field1
}
```

**Listing 2.27** CDS View with `Intersect`

**Example: Exception** If you want to determine those records that are only contained in one of two data sources, you can use the CDS syntax element `except`. CDS view `Z_ViewWithExcept` from [Listing 2.28](#) excepts records of CDS view `Z_UnionViewAsDataSourceB` from [Listing 2.25](#) from CDS view `Z_UnionViewAsDataSourceA` from [Listing 2.24](#). Because both these data sources contain a record with the value A for `Field1`, the result is defined by the record of CDS view `Z_UnionViewAsDataSourceA` with the value B for `Field1` (refer to [Table 2.5](#)).

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewWithExcept
  as select from Z_UnionViewAsDataSourceA
{
  key Field1
}
except select from Z_UnionViewAsDataSourceB
{
  key Field1
}
```

**Listing 2.28** CDS View with `Except`

## 2.13 Joins

*Joins* allow you to model conditional links between two data sources. The join conditions describe criteria for linking a data record of the primary data source with a data record of the secondary data source.

You can use elements of joined data sources for defining elements in the projection list of your CDS views. In addition, you can use the elements of joined data sources for enriching the `where` conditions of your CDS views.

Using joins



### Performance Aspects

Joins make optimizing the execution plan of the selection request at the database level more difficult. Therefore, you should only integrate those joins into the logic of your CDS view definition that are essentially required for the view's functionality, and avoid unnecessary joins of data sources resulting in denormalized CDS view models. This particularly applies to CDS views that you define for reuse purposes. Instead, you should consider providing appropriate associations between the CDS models. These associations can be used by the consumers of your model to enrich the data records in a convenient way, where this is actually necessary. We'll discuss associations further in [Chapter 3](#).

CDS views support four flavors of joins:

Types of joins

- **Left outer joins**  
These joins relate records of a primary data source with records of a secondary data source so that the result contains all the data records of the primary data source.
- **Right outer joins**  
These joins relate records of the secondary data source with the records



of the primary data source so that the result contains all the data records of the secondary data source.

- **Inner joins**

These joins relate records of a primary data source with records of a secondary data source so that the result contains only those records of the primary data source for which at least one join partner in the secondary data source exists.

- **Cross joins**

These joins combine all records of a primary data source with all records of a secondary data source. The number of records in the result set is equal to the number of records of the primary data source multiplied by the number of records of the secondary data source.

**Cardinality of joins** When dealing with joins, you'll often be confronted with the term *cardinality*. The cardinality of the join partner specifies the number of data records that result from the join relationship. For example, if there is more than one corresponding data record in the secondary data source for a single record of the primary data source, the number of resulting data records is multiplied by the cardinality of the join partner when applying the left outer join logic.

**Left outer and inner joins** In the following discussion, we'll focus on the left outer joins and inner joins, which are the most common join types. Sample CDS views `Z_ViewWithLeftOuterJoins` from [Listing 2.31](#) and `Z_ViewWithInnerJoins` from [Listing 2.33](#) (later in this section) will illustrate the various aspects of the join logic.

**Example: CDS views serving as data sources** Both these CDS views use CDS view `Z_ViewAsDataSourceD` from [Listing 2.29](#) as their primary data source.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewAsDataSourceD
  as select distinct from t000
{
  key cast( 'A' as abap.char(1) ) as FieldD1,
    cast( 'D' as abap.char(1) ) as FieldD2
}
union select distinct from t000
{
  key cast( 'C' as abap.char(1) ) as FieldD1,
    cast( 'E' as abap.char(1) ) as FieldD2
}
```

**Listing 2.29** CDS View `Z_ViewAsDataSourceD`

CDS view `Z_ViewAsDataSourceE` from [Listing 2.30](#) acts as their secondary data source.

```
@Metadata.ignorePropagatedAnnotations: true
define view entity Z_ViewAsDataSourceE
  as select distinct from t000
{
  key cast( 'D' as abap.char(1) ) as FieldE1,
  key cast( 'H' as abap.char(1) ) as FieldE2
}
union select distinct from t000
{
  key cast( 'D' as abap.char(1) ) as FieldE1,
  key cast( 'I' as abap.char(1) ) as FieldE2
}
union select distinct from t000
{
  key cast( 'F' as abap.char(1) ) as FieldE1,
  key cast( 'I' as abap.char(1) ) as FieldE2
}
```

**Listing 2.30** CDS View `Z_ViewAsDataSourceE`

CDS views `Z_ViewAsDataSourceD` and `Z_ViewAsDataSourceE` return two or three data records, as shown in [Table 2.6](#).

| CDS View                 | FieldD1 | FieldD2 | FieldE1 | FieldE2 |
|--------------------------|---------|---------|---------|---------|
| Z_ViewAsDataSourceD      | A       | D       | –       | –       |
|                          | C       | E       | –       | –       |
| Z_ViewAsDataSourceE      | –       | –       | D       | H       |
|                          | –       | –       | D       | I       |
|                          | –       | –       | F       | I       |
| Z_ViewWithLeftOuterJoins | A       | D       | –       | H       |
|                          | A       | D       | –       | I       |
|                          | C       | E       | –       | null    |
| Z_ViewWithInnerJoins     | A       | D       | –       | H       |
|                          | A       | D       | –       | I       |

**Table 2.6** Records of the Data Sources and the Join CDS Views Built on Top

**Example:** If these two CDS views are linked as data sources by a left outer join according to CDS view `Z_ViewWithLeftOuterJoins` from [Listing 2.31](#), the results list comprises three data records (see [Table 2.6](#)).

```
define view entity Z_ViewWithLeftOuterJoins
  as select from          Z_ViewAsDataSourceD
    left outer one to many join Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
  {
    key Z_ViewAsDataSourceD.FieldD1,
    key Z_ViewAsDataSourceD.FieldD2,
    key Z_ViewAsDataSourceE.FieldE2
  }
```

**Listing 2.31** CDS View with a Left Outer Join

**Example analysis** The first entry of CDS view `Z_ViewAsDataSourceD`, which has value A for key field `FieldD1`, is linked to two data records of CDS view `Z_ViewAsDataSourceE`. Consequently, the results list of CDS view `Z_ViewWithLeftOuterJoins` has two data records for this entry. This potential increase of the cardinality is expressed by addition to `many` in the left outer join statement. If you take a closer look at the values of `FieldE1` of the join partner CDS view `Z_ViewAsDataSourceE`, you'll recognize that there is no or a single (i.e., at maximum one) record of CDS view `Z_ViewAsDataSourceD` with a matching value of `FieldD1`. Accordingly, source cardinality of the left outer join in CDS view `Z_ViewWithLeftOuterJoins` is defined as `one`. Overall, this results in a one-to-many join.



### Specify the Cardinality of Joins

You should always specify the maximum target cardinality (to one or to exact one or to many) of a join partner. Furthermore, it may be useful to define the source cardinality (one or exact one or many) too. This specification is used both to document the composition of the CDS view and optimize the processing of a selection request in the database. However, if you maintain the cardinality information, you should make sure that it's defined correctly; otherwise, the selection result may become incorrect!

The second record of view `Z_ViewAsDataSourceD`, which has value C for key field `FieldD1`, doesn't have a join partner. According to the semantics of the left outer join, this data record remains in the results list. However, in this case, associated `FieldE2` of CDS view `Z_ViewWithLeftOuterJoins` receives value `null` on the database level.

If you transfer the selection result to an internal ABAP table according to [Listing 2.32](#), the null value is implicitly converted to the initial value of the corresponding ABAP field. By default, the ABAP runtime environment doesn't distinguish between initial values and null values in contrast to the CDS language and the SQL logic.

Null and initial values

```
SELECT *
  FROM z_viewwithleftouterjoins
 INTO TABLE @DATA(it_viewwithleftouterjoins).
```

**Listing 2.32** Data Selection from the CDS View from [Listing 2.31](#) in ABAP

### Required Handling of Null Values

On the SQL level, the difference between initial values and null values is significant in that null values represent nonvaluated data. In the CDS views, null values usually result from left outer joins without a matching join partner.

When modeling your CDS view logic, you must consider this distinction and handle it if necessary. For example, when comparing field values in where conditions, you have to consider potential null values. However, null values won't simply be replaced by initial values in your CDS logic. Otherwise, you may experience significant performance issues.



In contrast to a left outer join, an inner join removes source records without a join partner from the selection result.

Inner joins

Accordingly, the selection result of CDS view `Z_ViewWithInnerJoins` from [Listing 2.33](#) contains only two data records (refer to [Table 2.6](#)).

Example: CDS view with inner join

```
define view entity Z_ViewWithInnerJoins
  as select from Z_ViewAsDataSourceD
    inner join Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
  {
    key Z_ViewAsDataSourceD.FieldD1,
    key Z_ViewAsDataSourceD.FieldD2,
    key Z_ViewAsDataSourceE.FieldE2
  }
```

**Listing 2.33** CDS View with an Inner Join

You can achieve a corresponding selection result using a left outer join relationship too. In this case, the corresponding left outer join statement must be accompanied by additional where conditions, which ensure that a data

Left outer joins without null values

record of the respective join partner exists. Within these `where` conditions, you can validate that an adequate field (usually a key field) of the join partner has a value, which is different from the `null` value. [Listing 2.34](#) illustrates this approach by checking `FieldE2` of left outer join partner CDS view `Z_ViewAsDataSourceE`.

```
define view entity Z_ViewWithLeftOuterJoinsFiltrd
  as select from Z_ViewAsDataSourceD
    left outer one to many join Z_ViewAsDataSourceE
      on Z_ViewAsDataSourceD.FieldD2 = Z_ViewAsDataSourceE.FieldE1
  {
    key Z_ViewAsDataSourceD.FieldD1,
    key Z_ViewAsDataSourceD.FieldD2,
    key Z_ViewAsDataSourceE.FieldE2
  }
  where
    Z_ViewAsDataSourceE.FieldE2 is not null
```

**Listing 2.34** CDS View with a Left Outer Join and an Additional Existence Check for the Join Partner

#### Qualifying elements with their data sources

If you combine multiple data sources using joins, you must ensure that references to the elements of these data sources remain unique. To achieve this, you have to prefix the element name with the name of its data source, separated by a period. In the previous examples, this was realized by using the name of the data source as a qualifier. However, if the same data source is included multiple times in the same CDS view, you have to specify suitable alias names for each of its occurrences, as shown in [Listing 2.35](#). In this case, the CDS view `Z_ViewAsDataSourceD` is used twice as a data source. To distinguish between the two embeddings of this CDS view, they are assigned distinct alias names `D1` and `D2`.

```
define view entity Z_ViewWithJoinsAndDataSrcAlias
  as select from Z_ViewAsDataSourceD as D1
    left outer exact one to exact one join Z_ViewAsDataSourceD as D2
      on D1.FieldD1 = D2.FieldD1
  {
    key D1.FieldD1,
      D2.FieldD2
  }
}
```

**Listing 2.35** CDS View with Alias Names for the Same Data Source Joined Multiple Times

## 2.14 SQL Aggregation Functions

*SQL aggregation functions* allow you to perform calculations of predefined aggregates efficiently on the database level. You can use these SQL functions within the implementation of your CDS views.

To do this, first define the aggregation level to which you want to aggregate the result. Enter the aggregation level using syntax element `group by`, which should be placed after the projection list. It has to include all fields of the data sources that are included in the projection list of the CDS view. Fields whose aggregate is calculated are excluded from this. Within the projection list of the CDS view, you then apply the envisioned aggregation function to the fields that are relevant for aggregation.

An example is given by CDS view `Z_ViewAsDataSourceF` from [Listing 2.36](#) and CDS view `Z_ViewWithAggregations` from [Listing 2.37](#).

CDS view `Z_ViewAsDataSourceF` acts as the data source for the aggregation.

```
@Metadata.ignorePropagatedAnnotations: true
```

```
define view entity Z_ViewAsDataSourceF
  as select distinct from t000
  {
    key abap.char'A' as Field1,
    key abap.char'A' as Field2,
      abap.int1'1' as Field3
  }
  union all select distinct from t000
  {
    key abap.char'A' as Field1,
    key abap.char'B' as Field2,
      abap.int1'2' as Field3
  }
  union all select distinct from t000
  {
    key abap.char'A' as Field1,
    key abap.char'C' as Field2,
      abap.int1'3' as Field3
  }
}
```

**Listing 2.36** CDS View `Z_ViewAsDataSourceF`

It defines three data records, as depicted in [Table 2.7](#).

Define aggregations

Example:  
Aggregation

Example:  
CDS view serving  
as a data source

| Field1 | Field2 | Field3 |
|--------|--------|--------|
| A      | A      | 1      |
| A      | B      | 2      |
| A      | C      | 3      |

**Table 2.7** Records of CDS View Z\_ViewAsDataSourceF

**Example: Aggregating CDS view** CDS view Z\_ViewWithAggregations from Listing 2.37 contains the aggregation logic. Its group by statement defines that the data source records are to be aggregated up to the level of the first key field: Field1.

```
define view entity Z_ViewWithAggregations
  as select from Z_ViewAsDataSourceF
{
  key Field1,
  min(Field3)           as FieldWithMin,
  max(Field3)           as FieldWithMax,
  avg( Field3 as abap.decfloat34 ) as FieldWithAvg,
  cast( sum(Field3) as abap.int4 ) as FieldWithSum,
  count( distinct Field1 )   as FieldWithCountDistinct,
  count(*)               as FieldWithCountAll
}
group by Field1
```

**Listing 2.37** CDS View with Aggregation Functions

**Example analysis** The result of the selection from CDS view Z\_ViewWithAggregations comprises a single data record. The minimum (min) value (FieldWithMin = 1), maximum (max) value (FieldWithMax = 3), average (avg) value (FieldWithAvg = 2), and summed-up (sum) value (FieldWithSum = 6) of Field3 are calculated. In addition, for each entry in the results list of aggregating view Z\_ViewWithAggregations, the number (= 1) of its underlying aggregated original data records that differs in the value of grouping field Field1 (count distinct) is calculated and returned by FieldWithCountDistinct. Finally, the total number (= 3) of underlying aggregated data records (count(\*)) is calculated and returned by FieldWithCountAll.

**Aggregated field types** Note that the fields for minimum value FieldWithMin and maximum value FieldWithMax retain type int1 of underlying base field Field3. By default, this would also hold true for field FieldWithSum, which contains the result of the summation. However, in the example, a cast operation is applied, which explicitly extends the type to int4. When applying the average function, an

explicit type assignment is required. FieldWithAvg is typed as decfloat34. The fields that expose the result of the count operation—FieldWithCountDistinct and FieldWithCountAll—are implicitly assigned ABAP type int4.

### Avoid Memory Overflows

To avoid a memory overflow at the execution time of the data selection when using aggregation functions, you should check the expected value ranges and existing type assignments of the fields and enhance these types by explicit type conversions if necessary.

If you use aggregation functions, the grouping fields defined in the group by statement ideally should not contain any calculated or joined fields. Furthermore, it may be beneficial to perform the required aggregations as deep as possible within the CDS view stack to reduce the amount of processed data to its minimum as soon as possible.

If the values to be aggregated represent amounts or quantities, you must normalize them before the aggregation is performed. This means that you may first have to convert them to the same unit or keep the currency and unit information in the aggregation result to avoid unwanted calculation errors. We'll explain conversion functions in Section 2.18.

### Performance Aspects

If you use conversion functions on nonaggregated single data records, this can have a negative effect on performance. In such cases, you should consider first aggregating the data records while keeping their units and applying the conversion functions to the preaggregated data records.

## 2.15 Projection Fields

*Projection fields* are defined in the select list of a CDS view. Within the CDS view definition, you can access these fields by prefixing their names with operator \$projection followed by a dot. You can use projection fields to refer to calculation results defined within the same view. This allows you to build up nested calculations with the exposed fields serving as reusable intermediate results. In addition, you can use projection fields (with some restrictions) for defining associations (see Chapter 3).

Listing 2.38 illustrates the implementation of CDS view Z\_ViewWithProjectionFields, which leverages projection fields for further calculations.



Performance aspects

Using conversion functions



Example: CDS view with projection fields

```

define view entity Z_ViewWithProjectionFields
  as select distinct from t000
{
  abap.char'A' as FieldA,
  abap.char'B' as FieldB,
  concat( $projection.FieldA, $projection.FieldB ) as FieldC,
  concat( abap.char'A', abap.char'B' ) as FieldC2,
  concat( $projection.FieldC, abap.char'D' ) as FieldD,
  concat( concat( abap.char'A', abap.char'B' ), abap.char'D' )
    as FieldD2
}

```

Listing 2.38 CDS View with Projection Fields

**Example analysis** In this example, two fields, `FieldA` and `FieldB`, are defined with constant values A and B, respectively. These locally defined fields are concatenated to `FieldC`. The resulting value is the same as when concatenating the original constants, A and B; that is, `FieldC2` holds the same value as `FieldC`. Similarly, `FieldD` and `FieldD2` hold the same values, with `FieldD` reusing the determination logic of `FieldC` and with `FieldD2` calculating the result from scratch.

From a technical perspective, the modeled reuse on the CDS level is being expanded when creating the corresponding database view on SAP HANA. This is shown by the create statement in Listing 2.39, in which the definitions of fields `FieldC` and `FieldC2`—which are `FieldD` and `FieldD2`, respectively—are identical.

```

CREATE OR REPLACE VIEW "Z_VIEWWITHPROJECTIONFIELDS" AS SELECT
  DISTINCT N'A' AS "FIELDA",
  N'B' AS "FIELDB",
  CONCAT(
    N'A',
    N'B'
  ) AS "FIELDC",
  CONCAT(
    N'A',
    N'B'
  ) AS "FIELDC2",
  CONCAT(
    RTRIM(
      CONCAT(
        N'A',
        N'B'
      )
    ),

```

```

  N'D'
) AS "FIELD2",
CONCAT(
  RTRIM(
    CONCAT(
      N'A',
      N'B'
    )
  ),
  N'D'
) AS "FIELD2"
FROM "T000" "T000"

```

Listing 2.39 Create Statement of the Database View Generated from Listing 2.38

## 2.16 Parameters

*Parameters* are constituents of the CDS model signature. They represent scalar input values that must be supplied by the caller when performing data selections.

You can access and evaluate parameter values in the logic of your CDS models. This allows you to equip consumers of your CDS models with predefined control options for their data selections. You can also use parameters for enforcing a restriction of the selection result by applying them as predefined filter criteria.

Using parameters

Parameters are listed directly after the name of the CDS model. The parameter list is introduced by CDS syntax element `with parameters`. Each parameter has a name and a type separated by a colon. The type can be based on elementary ABAP types, data elements, and simple types.

Definition of parameters

The example from Listing 2.40 shows two alternatives for assigning types to parameters.

```

define view entity Z_ViewWithParameters
  with parameters
    P_KeyDate : abap.dats,
    P_Language : sylanu
  as select from ...

```

Listing 2.40 Definition of Parameters

The parameter names must be unique within a CDS model and must be different from the element names.



### Parameter Names

You should start the name of parameters with the prefix `P_`, followed by the semantic name of the transported value. Following this rule creates a clear separation of the parameter names from the names of the remaining elements of the CDS models.

**Parameters in CDS views** Parameters can be used in different places within a CDS model. For example, you can use parameters to do the following:

- Define where conditions in a CDS view.
- Calculate fields in the projection list of a CDS view.
- Supply parameters of other data sources with the requested values.

Within the implementation of a CDS model, parameters can be accessed by the preceding syntax element `$parameters`, which is separated by a dot from the parameter name.

Let's now have a look at the potential usages of parameters with the following sample CDS views and ABAP code snippets.

**Example: CDS view serving as a data source** In all depicted cases, CDS view `Z_ViewWithParametersDataSource` from [Listing 2.41](#) serves as the fundamental data source. According to its definition, it returns a single data record.

```
define view entity Z_ViewWithParametersDataSource
  as select distinct from t000
{
  key abap.char'A'      as KeyField,
  key abap.lang'E'     as Language,
  key abap.dats'99580809' as ValidityEndDate,
  abap.dats'20111004' as ValidityStartDate
}
```

**Listing 2.41** CDS View `Z_ViewWithParametersDataSource`

**Example: CDS view with parameters** CDS view `Z_ViewWithParameters` in [Listing 2.42](#) uses CDS view `Z_ViewWithParametersDataSource` as its direct base data source. It defines parameters for key date `P_KeyDate` and language `P_Language`.

```
define view entity Z_ViewWithParameters
  with parameters
    P_KeyDate : abap.dats,
    P_Language : sylangu
  as select from Z_ViewWithParametersDataSource
```

```
association [0..*] to Z_ViewWithParametersAscTarget as _Target
  on $projection.KeyField = _Target.KeyField
association [0..1] to Z_ViewWithParametersAscTarget as _FilteredTarget
  on $projection.KeyField = _FilteredTarget.KeyField
  and $projection.Language = _FilteredTarget.Language
{
  key KeyField,
  ValidityEndDate,
  ValidityStartDate,
  $parameters.P_Language as Language,
  _Target(P_ValidityDate: $parameters.P_KeyDate)
  [1:Language= $parameters.P_Language].KeyField as TargetKeyField,
  _FilteredTarget
}
where ValidityEndDate >= $parameters.P_KeyDate
  and ValidityStartDate <= $parameters.P_KeyDate
  and Language = $parameters.P_Language
```

**Listing 2.42** CDS View with Parameters

These parameters are used in its where condition for restricting the selection result. Parameter `P_Language` is also added as field `Language` to the projection list of the CDS view. This field will have the same value as the parameter for all data records. As a result, the parameter value can indirectly be used in the definition of association `_FilteredTarget` in expression `$projection.Language = _FilteredTarget.Language`. Alternatively, you could use the parameter directly on the right-hand side of an expression such as `_FilteredTarget.Language = $parameters.P_Language`.

The target of association `_FilteredTarget` is CDS view `Z_ViewWithParametersAscTarget` with parameter `P_ValidityDate`, as shown in [Listing 2.43](#).

```
define view entity Z_ViewWithParametersAscTarget
  with parameters
    P_ValidityDate : abap.dats
  as select from Z_ViewWithParametersDataSource
{
  key KeyField,
  key Language,
  ValidityEndDate,
  ValidityStartDate
}
where ValidityEndDate >= $parameters.P_ValidityDate
  and ValidityStartDate <= $parameters.P_ValidityDate
```

**Listing 2.43** CDS View with a Parameter Serving as an Association Target

**Example analysis**

**Example: Association target with parameter**

**Parameters in path expressions** When accessing field `KeyField` of this target CDS view by applying path expression `_Target(P_ValidityDate: $parameters.P_KeyDate)[1:Language=$parameters.P_Language].KeyField` in CDS view `Z_ViewWithParameters`, its own parameter `P_ValidityDate` must be supplied. This is achieved by binding it to parameter `P_KeyDate` of CDS view `Z_ViewWithParameters`.

Within the depicted path expression, the second local parameter `P_Language` of the source view `Z_ViewWithParameters` is also used for applying a filter condition on associated target field `Language`. This means that the key and thus the associated data record of target CDS view `Z_ViewWithParametersAscTarget` are uniquely identified. As a result, the maximum cardinality of the path expression is defined as 1.



### Missing Support for Specifying Target Parameters in Association Definitions

Parameters of association targets can't be supplied within the definition of an association. Among other things, this means you can't explicitly bind parameters of associated CDS models. Therefore, a coupling of the parameters in CDS models can't be enforced. Instead, the consumer has to supply the parameters consistently when selecting data from the associated CDS models.

**Example:**  
Parameters in another CDS view

If you want to use CDS model `Z_ViewWithParameters` as a data source within a select statement, its parameters must be supplied with values. For example, [Listing 2.44](#) shows CDS view `Z_ViewWithParametersConsumer`, which supplies parameters `P_KeyDate` and `P_Language` of its data source with session variable `$session.system_date` and constant value `E`, respectively.

```
define view entity Z_ViewWithParametersConsumer
  as select from Z_ViewWithParameters(
    P_KeyDate: $session.system_date,
    P_Language: 'E')
{
  key KeyField
}
```

**Listing 2.44** CDS View Supplying Parameters of Its Data Source

**Example:**  
Parameters in ABAP

[Listing 2.45](#) shows a corresponding selection in ABAP code. In this example, ABAP system field `sy-datum` is used for supplying parameter `P_KeyDate`.

```
SELECT keyfield
  FROM z_viewwithparameters(
    p_keydate = @sy-datum,
```

```
    p_language = 'E' )
  INTO TABLE @DATA(lt_viewwithparameters).
```

**Listing 2.45** Data Selection in ABAP Corresponding to the CDS Modeling in [Listing 2.44](#)

Within the CDS view stack, parameters must always be supplied. In contrast, the ABAP SQL interface supports the automatic supply of parameters, which are annotated with `@Environment.systemField...`. The permitted values of this annotation, such as `SYSTEM_DATE` and `SYSTEM_LANGUAGE`, correspond to the values of the ABAP system fields, such as `sy-datum` and `sy-langu`. If such an annotated parameter isn't supplied explicitly with a value when selecting data via the ABAP SQL interface, the ABAP runtime automatically fills the parameter with the value of the corresponding ABAP system field. As a result, the corresponding parameter of the CDS model becomes optional from an ABAP perspective.

This is demonstrated by the following example, which comprises the CDS view `Z_ViewWithOptionalParameters` in [Listing 2.46](#) and the two data selections in [Listing 2.47](#) and [Listing 2.48](#).

```
define view entity Z_ViewWithOptionalParameters
  with parameters
    @Environment.systemField: #SYSTEM_DATE
    P_KeyDate : abap.dats
  as select distinct from Z_ViewWithParametersDataSource
{
  key KeyField
}
where ValidityEndDate >= $parameters.P_KeyDate
  and ValidityStartDate <= $parameters.P_KeyDate
```

**Listing 2.46** CDS View with a Specially Annotated Parameter

Parameter `P_KeyDate` of CDS view `Z_ViewWithOptionalParameters` is annotated with `@Environment.systemField: #SYSTEM_DATE`. It's explicitly supplied in [Listing 2.47](#) with system field `sy-datum`.

```
SELECT *
  FROM z_viewwithoptionalparameters( p_keydate = @sy-datum )
  INTO TABLE @DATA(lt_viewwithoptionalparameters).
```

**Listing 2.47** Data Selection in ABAP with Explicit Supply of the CDS View Parameter

**Mandatory and optional supply of parameter values**

**Example:**  
Data selection in ABAP

In contrast, the data selection in [Listing 2.48](#) doesn't supply this parameter explicitly.

```
SELECT *
  FROM z_viewwithoptionalparameters
  INTO TABLE @DATA(lt_viewwithoptionalparameters).
```

**Listing 2.48** Data Selection in ABAP with Implicit Supply of the CDS View Parameter

Nevertheless, both selections yield the same result.



### Check Usage of Parameters

Because parameters typically force consumers of a CDS model to supply them with values, the consumers must know the lists of permitted values to carry out data selections successfully. Therefore, a simple data selection may not be possible without this knowledge. Furthermore, the addition, modification, or removal of a parameter is usually incompatible for consumers of the affected CDS model because consumers have to react to these changes. Therefore, you should only define parameters in your CDS models if they are essential for the functionality of the corresponding CDS models.

## 2.17 Reference Fields

*Reference fields* are amount and quantity fields that define references to currency and unit fields, respectively. Such a reference is technically enforced for fields that have the ABAP type `curr` or `quan`.

**Defining amount and quantity fields**

In CDS models, references to currencies and units are established by means of annotations `@Semantics.amount.currencyCode...` and `@Semantics.quantity.unitOfMeasure...`, respectively. Therein, the assigned annotation values represent the respective currency and unit fields that the annotated fields refer to. The referenced currency and unit fields with fields of type `curr` and `quan` have to be defined as fields of type `cuky` or `unit`. The currency codes themselves originate from table `TCURC`. The quantity units are defined by table `T006`.

**Example:**  
CDS view with amount and quantity fields

[Listing 2.49](#) shows sample CDS view `Z_ViewWithReferenceFieldsA` with quantity and amount fields.

```
define view entity Z_ViewWithReferenceFieldsA
  as select distinct from t000
{
```

```
@Semantics.quantity.unitOfMeasure: 'QuantityUnit'
abap.quan '1234.56'                as Quantity,
abap.unit 'PC'                    as QuantityUnit,
@Semantics.amount.currencyCode: 'Currency'
abap.curr '1234.56'               as Amount,
abap.cuky 'USD'                   as Currency
}
```

**Listing 2.49** CDS View with Amount and Quantity Fields

### Use Curr Fields with Two Decimals Only

When defining `curr` fields, you should always define them with two decimals. Otherwise, the currency shift logic may not work properly.

You'll learn more about the currency shift logic in [Chapter 17, Section 17.2.3](#).

Besides plain quantities referring to units of table `T006`, the CDS models also support defining quantities that are calculated from other quantities and amounts. For regular quantities, for example, the units of such calculated quantities are referenced by `@Semantics.quantity.unitOfMeasure...`. These units can represent complex expressions involving various base units and currencies. Accordingly, they are defined as character fields. Typically, such fields are modeled as virtual fields in the CDS models and populated by ABAP code. Of course, if technically feasible, you could also construct the unit values by applying suitable string operations, as depicted earlier in [Listing 2.46](#).

**Calculated quantities**

If a `curr` field will be incorporated into a calculation, it first must be converted into a plain value without its currency reference by using function `get_numeric_value`. Similarly, if you want to decouple quantities from their unit, you can use the same function. For a `curr` field, this function not only removes the reference and converts the `curr` value to a value of type `decfloat34` but also applies an implicit decimal shift.

**Type conversions of reference fields**

If you only want to convert the type of an amount field from `curr` to `decfloat` while keeping its reference field nature, you can use function `curr_to_decfloat_amount`. Because the resulting field is to be considered an amount, it has to specify a reference to its currency via `@Semantics.amount.currencyCode...`

CDS view `Z_ViewWithReferenceFieldsB` from [Listing 2.50](#), which is based on CDS view `Z_ViewWithReferenceFieldsA` from [Listing 2.49](#), illustrates the usage of the aforementioned conversion functions and the definition of a calculated quantity, including its unit.

**Example:**  
CDS view with type conversions of reference fields



```

define view entity Z_ViewWithReferenceFieldsB
  as select from Z_ViewWithReferenceFieldsA
  {
    @Semantics.quantity.unitOfMeasure: 'QuantityUnit'
    Quantity,
    QuantityUnit,
    @Semantics.amount.currencyCode: 'Currency'
    Amount,
    Currency,
    get_numeric_value( Amount )      as AmountWithoutReference,
    get_numeric_value( Quantity )    as QuantityWithoutReference,
    @Semantics.amount.currencyCode: 'Currency'
    curr_to_decfloat_amount( Amount ) as DecfloatAmount,
    @Semantics.quantity.unitOfMeasure: 'CalculatedUnit'
    get_numeric_value( Amount ) / $projection.quantity
                                   as AmountPerQuantity,

    cast( concat( Currency,
                 concat( '/', QuantityUnit)
               ) as abap.char(50) ) as CalculatedUnit
  }

```

Listing 2.50 CDS View with Converted Amount and Quantity Fields

## Display reference information

If you want to get an overview about the reference fields of your CDS entities, you can place the cursor on the CDS entity name and press **F2**. This will open up a popup that lists the fields along with their types and references, as illustrated in [Figure 2.2](#) for CDS view `Z_ViewWithReferenceFieldsB` from [Listing 2.50](#).

| Column                   | Data Element           | Data Type      | Unit/Currency Reference | Description             |
|--------------------------|------------------------|----------------|-------------------------|-------------------------|
| Quantity                 |                        | quan(6,2)      | quantityunit            |                         |
| QuantityUnit             |                        | unit(2)        |                         |                         |
| Amount                   |                        | curr(6,2)      | currency                |                         |
| Currency                 |                        | cuky(5)        |                         |                         |
| AmountWithoutReference   |                        | decfloat34(34) |                         |                         |
| QuantityWithoutReference |                        | decfloat34(34) |                         |                         |
| DecfloatAmount           |                        | decfloat34(34) | currency                |                         |
| AmountPerQuantity        |                        | decfloat34(34) | calculatedunit          |                         |
| CalculatedUnit           | dd_cds_calculated_unit | char(50)       |                         | DD: CDS Calculated Unit |

Figure 2.2 Technical Field Information of the CDS View from [Listing 2.50](#)

## 2.18 Conversion Functions for Currencies and Quantity Units

*Conversion functions* for the conversion of currencies and units of measure are based on several sets of persistent data records. These data records are included in the evaluation logic during processing of the conversion logic. The corresponding persistency captures, for example, the time-dependent conversion factors between currencies of different countries. This information must be integrated into the conversion logic based on the relevant key date.

The CDS syntax allows you to embed conversion functions directly into the implementation of your CDS views.

[Listing 2.51](#) depicts such an embodiment for a unit conversion.

```

define view entity Z_ViewWithUnitConversions
  with parameters
    P_DisplayUnit : msehi
  as select from ZI_SalesOrderItem
  {
    key SalesOrder,
    key SalesOrderItem,
    @Semantics.quantity.unitOfMeasure: 'OrderQuantityUnit'
    OrderQuantity,
    OrderQuantityUnit,
    @Semantics.quantity.unitOfMeasure: 'OrderQuantityDisplayUnit'
    unit_conversion( quantity      => OrderQuantity,
                    source_unit    => OrderQuantityUnit,
                    target_unit    => $parameters.P_DisplayUnit,
                    error_handling => 'FAIL_ON_ERROR' )
                    as OrderQuantityInDisplayUnit,
    $parameters.P_DisplayUnit as OrderQuantityDisplayUnit
  }

```

Listing 2.51 CDS View with Unit Conversion

[Listing 2.52](#) shows a corresponding example of a currency conversion.

```

define view entity Z_ViewWithCurrencyConversions
  with parameters
    P_DisplayCurrency : waers_curc,
    P_ExchangeRateDate : sydatum
  as select from ZI_SalesOrderItem
  {
    key SalesOrder,

```

Conversion functions in CDS views

Example: Unit conversion

Example: Currency conversion

```

key SalesOrderItem,
@Semantics.amount.currencyCode: 'TransactionCurrency'
NetAmount,
TransactionCurrency,
@Semantics.amount.currencyCode: 'DisplayCurrency'
currency_conversion(
    amount          => NetAmount,
    source_currency => TransactionCurrency,
    target_currency => $parameters.P_DisplayCurrency,
    exchange_rate_date => $parameters.P_ExchangeRateDate,
    exchange_rate_type => 'M',
    round           => 'X',
    decimal_shift   => 'X',
    decimal_shift_back => 'X',
    error_handling  => 'SET_TO_NULL' )
as NetAmountInDisplayCurrency,
$parameters.P_DisplayCurrency as DisplayCurrency
}

```

Listing 2.52 CDS View with Currency Conversion

**Analysis of the examples**

The conversion functions have mandatory input parameters as well as optional parameters. The latter only need to be supplied if you want to override the default values that are assigned to them automatically. Depending on the concrete input parameter of the chosen conversion function, you can supply it with a literal value, with an actual value from the respective data record, or with a parameter value.

**Unit conversion**

The unit conversion uses function `unit_conversion`. Earlier in [Listing 2.47](#), the quantity (`quantity`) and its unit (`source_unit`) are provided as input parameters to the conversion function. These input parameters are bound to fields `OrderQuantity` and `OrderQuantityUnit` of CDS view `Z_ViewWithUnitConversions`. The semantic correlation between these two fields is expressed by annotation `@Semantics.quantity.unitOfMeasure: 'OrderQuantityUnit'` of field `OrderQuantity`.

The target unit of conversion `target_unit` is defined by CDS parameter `P_DisplayUnit`. The result of the conversion is returned via field `OrderQuantityInDisplayUnit`. This field is related to unit field `OrderQuantityDisplayUnit`, which is derived from CDS parameter `P_DisplayUnit` via its annotation `@Semantics.quantity.unitOfMeasure: 'OrderQuantityDisplayUnit'`.

**Currency conversion**

The currency conversion (`currency_conversion`) from [Listing 2.48](#) essentially corresponds to the previously discussed unit conversion. However, more input parameters are included in the calculation. Besides the amount

(`amount`), which is filled with the actual value of field `NetAmount`, the currency (source\_currency), which is filled with the actual value of field `TransactionCurrency`, and target currency (`target_currency`), which is specified by CDS parameter `P_DisplayCurrency`, in particular, the key date of conversion `exchange_rate_date` must be specified by the function's caller. In our case, this is specified by CDS parameter `P_ExchangeRateDate`. Furthermore, the exchange rate type (`exchange_rate_type`) is set to fixed value `M`.

The currency conversion function allows you to activate or deactivate the business rounding (`round`) and the usage of shifts of decimal places before (`decimal_shift`) and after the calculation (`decimal_shift_back`). In the example described here, all these parameters are filled with constant literal value `X`. This valuation corresponds to the default values of the supplied parameters.

The relationships between the amount fields and their currency fields are expressed by annotations `@Semantics.amount.currencyCode: ...`

Particular attention must be paid to the handling of possible errors when dealing with conversion functions. The two examples just described apply error handling `FAIL_ON_ERROR`. This corresponds to the default value of the corresponding input parameter and causes a runtime error in the database processing if the conversion can't be carried out successfully.

Error handling

There are many possible root causes for such errors. For example, the unit conversion can't be executed successfully if the value of the target unit, which is supplied by the related input parameter, doesn't exist. Error handling `FAIL_ON_ERROR` therefore requires a high degree of data consistency and data completeness, as well as its own prevalidations of the supplied input parameters, for keeping possible error situations to a minimum.

Besides error-handling value `FAIL_ON_ERROR`, the conversion functions also support error-handling values `KEEP_UNCONVERTED` and `SET_TO_NULL`. Value `KEEP_UNCONVERTED` can be used for preserving the original value as the target value if problems occur in the context of the conversion. Value `SET_TO_NULL` can be used for setting the target value to `null` in case of issues. If you require any of these error-handling strategies, you should define the corresponding input parameter of the conversion function accordingly. However, note that you might not be able to detect issues easily if you choose an error-handling strategy that is different from `FAIL_ON_ERROR`.

**Conversion Functions in Analytical Queries**

In analytical query CDS views, the conversion functions for currencies and quantities are performed by the analytic engine after the records are aggregated. This can significantly improve the performance compared to



applying the same conversion functions already in the underlying data sources.

In addition, the analytic engine implements its own handling for conversion errors: when applying the conversion in an analytical query view, the original value with its unit is preserved if the requested conversion fails.

You'll find more information about analytical queries and the analytical engine in [Chapter 10, Section 10.3](#).

## 2.19 Provider Contracts

### Background information

*CDS provider contracts* formally define rules that the definition of a CDS model, which is classified accordingly, must follow. The rules consider pre-defined usage scenarios of the CDS models. Content-wise, the provider contracts correspond to the modeling patterns captured by annotation `@ObjectModel.modelingPattern`. Within a composition hierarchy of CDS models, only the root CDS model may be assigned a provider contract. Compositional children inherit the provider contract from their root.

The underlying rules of the provider contracts will ensure that CDS entities can actually be used as specified. They may result in both restricting the admissible modeling options and enabling framework-specific functions. Major parts of the rules are enforced by the CDS syntax check and as such may impact the activation of the CDS models, which means if you assign a provider contract to a CDS model but contradict its rules, it may not be possible to activate the CDS model. In contrast to the provider contract being defined by the corresponding CDS syntax elements, the aforementioned annotation `@ObjectModel.modelingPattern` doesn't influence the activation but serves documentation purposes only.

### Overview

[Table 2.8](#) presents an overview of the provider contracts and their intended usages.

| Provider Contract       | Usage                                                                                                                                                                                                                                         |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| transactional_interface | This CDS projection view serves as a programmatic and modeled interface of the functionality of business objects in the ABAP RESTful application programming model context. Only simple projections of the underlying entities are supported. |

**Table 2.8** Provider Contracts

| Provider Contract   | Usage                                                                                                                                                                                                                                                                                                                                 |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| transactional_query | This CDS projection view defines a service interface on top of a CDS view model for transactional processing. CDS projection views with this provider contract may be defined on top of regular CDS views without a provider contract or on top of CDS projection views with provider contract <code>transactional_interface</code> . |
| analytical_query    | This transient query CDS projection view is executed by the analytical infrastructure.                                                                                                                                                                                                                                                |

**Table 2.8** Provider Contracts (Cont.)

[Listing 2.53](#) to [Listing 2.55](#) demonstrate an example for the usage of the provider contracts in a CDS view stack. Fundamental CDS view `Z_ViewWithProviderContractA` from [Listing 2.53](#) acts as the root CDS view of a compositional hierarchy.

```
define root view entity Z_ViewWithProviderContractA
  as select distinct from t000
{
  key abap.char'A' as KeyField
}
```

**Listing 2.53** CDS View `Z_ViewWithProviderContractA`

CDS view `Z_ViewWithProviderContractB` from [Listing 2.54](#) leverages CDS view `Z_ViewWithProviderContractA` as its data source. It's defined as a root CDS projection view with provider contract `transactional_interface` via the corresponding syntax element. In addition, this CDS view is annotated with `@ObjectModel.modelingPattern: #TRANSACTIONAL_INTERFACE`.

```
@ObjectModel.modelingPattern: #TRANSACTIONAL_INTERFACE
define root view entity Z_ViewWithProviderContractB
  provider contract transactional_interface
  as projection on Z_ViewWithProviderContractA
{
  key KeyField
}
```

**Listing 2.54** CDS Projection View `Z_ViewWithProviderContractB`

CDS view `Z_ViewWithProviderContractB` itself is used as a data source of the transactional query CDS view `Z_ViewWithProviderContractC` from [Listing](#)

**Example:**  
CDS views with  
provider contracts

2.55, which is assigned CDS provider contract `transactional_query` and is annotated accordingly too.

```
@ObjectModel.modelingPattern: #TRANSACTIONAL_QUERY
define root view entity Z_ViewWithProviderContractC
  provider contract transactional_query
  as projection on Z_ViewWithProviderContractB
{
  key KeyField
}
```

**Listing 2.55** CDS Projection View `Z_ViewWithProviderContractC`

## 2.20 Entity Buffer Definitions

**Use cases** *Entity buffer definitions* allow you to speed up selections from CDS views in ABAP. This is achieved by processing selection requests directly on the ABAP application server instead of delegating them to the SAP HANA database.

**Bufferable CDS views** To be able to buffer a CDS view, the implemented logic of the CDS view is subject to several restrictions. For example, CDS views with parameters and CDS views that use functions with varying results such as `utcl_current` can't be buffered. Those responsible for the CDS views need to document by means of annotation `@AbapCatalog.entityBuffer.definitionAllowed:true` that they are aware of these technical limitations and that they will consider these restrictions during the further lifecycle of the annotated CDS views.

**Example: CDS view with buffer option** [Listing 2.56](#) shows CDS view `Z_ViewWithBufferA`, which is annotated accordingly. It selects directly from table `T000`.

```
@AbapCatalog.entityBuffer.definitionAllowed: true
define view entity Z_ViewWithBufferA as select from t000
{
  key mandt as KeyField,
  mtext as Field
}
```

**Listing 2.56** CDS View `Z_ViewWithBufferA` with Buffer Option

**Buffering of stacked CDS views** If you want to enable a buffer for a CDS view, which uses another CDS view as its data source, not only the CDS view itself but also all the underlying CDS views must support buffers in principle.

CDS view `Z_ViewWithBufferB` from [Listing 2.57](#) also supports buffering. This is technically feasible because its own data source—CDS view `Z_ViewWithBufferA`—also offers a buffer option.

**Example:** CDS view hierarchy with buffer option

```
@AbapCatalog.entityBuffer.definitionAllowed: true
define view entity Z_ViewWithBufferB as select from Z_ViewWithBufferA
{
  key KeyField,
  Field
}
```

**Listing 2.57** CDS View `Z_ViewWithBufferB` with Buffer Option

We now want to create an entity buffer definition. Entity buffer definitions can be created from within the ADT **Project Explorer** view. Therein, select the CDS view to be buffered and call function **New Entity Buffer** from its context menu. Enter the requested information in the dialog that is launched and confirm the creation.

**Create entity buffer definitions**

In our case, we want to create a buffer entity definition for CDS view `Z_ViewWithBufferA` from [Listing 2.56](#) that is named equally. This entity buffer definition will be assigned to the lowest definition layer `core` and specify a buffer strategy on a single record level (type `single`). [Listing 2.58](#) demonstrates the corresponding definition.

**Example:** Entity buffer definition

```
define view entity buffer on Z_ViewWithBufferA
  layer core
  type single
```

**Listing 2.58** Entity Buffer Definition for CDS View `Z_ViewWithBufferA` in the Layer Core

There are multiple logical layers to which buffer definitions can be assigned. This allows specifying multiple entity buffer definitions for a single CDS view. However, in a single layer, there may at maximum be a single entity buffer definition per CDS view. The entity buffer definition of the highest layer determines the effective buffer handling.

**Layering of entity buffer definitions**

Imagine that CDS view `Z_ViewWithBufferA` from [Listing 2.56](#) and its buffer definition from [Listing 2.58](#) could not be redefined (e.g., because they were shipped by SAP), but you would like to disable the buffer. You can achieve this by creating your own entity buffer definition for CDS view `Z_ViewWithBufferA` in a higher layer and deactivate the standard buffering from there. [Listing 2.59](#) illustrates the corresponding entity buffer definition `ZZ1_ViewWithBufferA`.

**Example:** Overruling buffer definitions

```
define view entity buffer on Z_ViewWithBufferA
  layer customer
  type none
```

**Listing 2.59** Entity Buffer Definition for CDS View Z\_ViewWithBufferA in Layer Customer

It's assigned the highest layer `customer` and specifies via the declaration `type none` that no buffering will be applied. This setting overrules the entity buffer definition from [Listing 2.58](#).



### Buffer Handling

The definition and application of buffers will always be aligned with the actual data selections: on one side, buffers may restrict the further evolution of the buffered CDS views, and on the other side, you have to be aware that even if a buffer exists, not all selections can be executed on the ABAP application server. Furthermore, the administration of the buffers can imply a significant overhead and negatively impact resource consumption and performance. This is especially true if a huge amount of data is buffered, if the control settings of the buffers aren't chosen in an optimal way, or if the buffered data is invalidated frequently. You should also consider that entity buffers may be built up in parallel to existing table buffers. In such cases, you should evaluate whether both buffers are actually required or whether you could switch your selections for leveraging a single buffer only.

## 2.21 Summary

In this chapter, you learned how data provisioning can be implemented by leveraging SQL concepts such as joins, unions, and aggregation functions, as well as specialized CDS functions such as entity buffers. You also were introduced to CDS parameters. CDS parameters can be applied for enforcing user input as well as for providing context information that can be used for implementing selection logic.

In the next chapter, you'll learn more about associations, which were only briefly introduced in this chapter.

# Contents

|                                                              |           |
|--------------------------------------------------------------|-----------|
| Preface .....                                                | 17        |
| <b>1 Modeling Your First CDS Views</b> .....                 | <b>23</b> |
| <b>1.1 Define the Data Model of the Application</b> .....    | 24        |
| <b>1.2 Implement the Data Model of the Application</b> ..... | 26        |
| 1.2.1 Create Database Tables .....                           | 28        |
| 1.2.2 Create a CDS View .....                                | 31        |
| 1.2.3 Edit a CDS View .....                                  | 37        |
| 1.2.4 Create a Hierarchy of CDS Views .....                  | 40        |
| <b>1.3 Summary</b> .....                                     | 52        |
| <b>2 Fundamentals of CDS Data Modeling</b> .....             | <b>53</b> |
| <b>2.1 Overview of CDS Models</b> .....                      | 54        |
| <b>2.2 Overview of CDS View Syntax</b> .....                 | 58        |
| <b>2.3 Key Fields</b> .....                                  | 61        |
| <b>2.4 Cast Operations</b> .....                             | 62        |
| <b>2.5 Typed Literals</b> .....                              | 64        |
| <b>2.6 Simple Types</b> .....                                | 66        |
| <b>2.7 Case Statements</b> .....                             | 68        |
| <b>2.8 Session Variables</b> .....                           | 69        |
| <b>2.9 Client Handling</b> .....                             | 71        |
| <b>2.10 Select Distinct Statements</b> .....                 | 72        |
| <b>2.11 Union Views</b> .....                                | 73        |
| 2.11.1 Union Definitions .....                               | 73        |
| 2.11.2 Union and Union All Logic .....                       | 78        |
| <b>2.12 Intersect and Except Statements</b> .....            | 79        |
| <b>2.13 Joins</b> .....                                      | 81        |
| <b>2.14 SQL Aggregation Functions</b> .....                  | 87        |

|             |                                                               |     |
|-------------|---------------------------------------------------------------|-----|
| <b>2.15</b> | <b>Projection Fields</b>                                      | 89  |
| <b>2.16</b> | <b>Parameters</b>                                             | 91  |
| <b>2.17</b> | <b>Reference Fields</b>                                       | 96  |
| <b>2.18</b> | <b>Conversion Functions for Currencies and Quantity Units</b> | 99  |
| <b>2.19</b> | <b>Provider Contracts</b>                                     | 102 |
| <b>2.20</b> | <b>Entity Buffer Definitions</b>                              | 104 |
| <b>2.21</b> | <b>Summary</b>                                                | 106 |

### **3 Associations** 107

|            |                                                     |     |
|------------|-----------------------------------------------------|-----|
| <b>3.1</b> | <b>Define Associations</b>                          | 108 |
| <b>3.2</b> | <b>Expose Associations</b>                          | 111 |
| <b>3.3</b> | <b>Model Compositional Relations</b>                | 111 |
| <b>3.4</b> | <b>Model M:N Relations</b>                          | 114 |
| <b>3.5</b> | <b>Project Associations</b>                         | 117 |
| <b>3.6</b> | <b>Use Associations in CDS Views</b>                | 118 |
| 3.6.1      | Define Path Expressions                             | 118 |
| 3.6.2      | Implicit Joins                                      | 121 |
| 3.6.3      | Cardinality Changes Resulting from Path Expressions | 124 |
| 3.6.4      | Restrictions for Defining Path Expressions          | 127 |
| <b>3.7</b> | <b>Use Associations in ABAP Code</b>                | 129 |
| <b>3.8</b> | <b>Summary</b>                                      | 129 |

### **4 Annotations** 131

|            |                               |     |
|------------|-------------------------------|-----|
| <b>4.1</b> | <b>Annotation Definitions</b> | 132 |
| 4.1.1      | Syntax Overview               | 132 |
| 4.1.2      | Annotation Names              | 136 |
| 4.1.3      | Annotation Types and Values   | 139 |
| 4.1.4      | Enumeration Values            | 140 |
| 4.1.5      | Annotation Default Values     | 141 |
| 4.1.6      | Annotation Scopes             | 142 |
| <b>4.2</b> | <b>Effects of Annotations</b> | 143 |

|            |                                                  |     |
|------------|--------------------------------------------------|-----|
| <b>4.3</b> | <b>Propagation Logic for Annotations</b>         | 145 |
| 4.3.1      | Propagation Logic within Simple Type Hierarchies | 145 |
| 4.3.2      | Propagation Logic of Element Annotations         | 147 |
| 4.3.3      | Consistency Aspects                              | 152 |
| <b>4.4</b> | <b>Metadata Extensions</b>                       | 155 |
| <b>4.5</b> | <b>Active Annotations</b>                        | 158 |
| <b>4.6</b> | <b>Summary</b>                                   | 160 |

### **5 Access Controls** 161

|            |                                                                       |     |
|------------|-----------------------------------------------------------------------|-----|
| <b>5.1</b> | <b>Fundamentals of Access Controls</b>                                | 162 |
| <b>5.2</b> | <b>Mode of Action of Access Controls</b>                              | 166 |
| <b>5.3</b> | <b>Implementation Patterns for Access Controls</b>                    | 171 |
| 5.3.1      | Implement Access Controls with Path Expressions                       | 171 |
| 5.3.2      | Inherit Implementation of Access Controls                             | 180 |
| 5.3.3      | Implement Access Controls without Using Authorization Objects         | 189 |
| 5.3.4      | Implement Access Controls for Analytical Queries                      | 194 |
| 5.3.5      | Implement Access Controls for Transactional Applications              | 196 |
| 5.3.6      | Implement Access Controls on the Field Level                          | 199 |
| 5.3.7      | Change Access Controls of SAP-Delivered CDS Models                    | 200 |
| 5.3.8      | Block Standard Data Selections from CDS Models                        | 203 |
| 5.3.9      | Decouple Access Controls from User Input                              | 205 |
| 5.3.10     | Map CDS Fields onto Fields of Authorization Objects Using Indirection | 207 |
| <b>5.4</b> | <b>Test Access Controls</b>                                           | 207 |
| <b>5.5</b> | <b>Summary</b>                                                        | 210 |

### **6 Business Services** 211

|            |                            |     |
|------------|----------------------------|-----|
| <b>6.1</b> | <b>Projection Views</b>    | 212 |
| <b>6.2</b> | <b>Service Definitions</b> | 216 |
| <b>6.3</b> | <b>Service Bindings</b>    | 221 |
| 6.3.1      | OData UI Services          | 222 |

|            |                                                      |            |
|------------|------------------------------------------------------|------------|
| 6.3.2      | OData Web API Services .....                         | 227        |
| 6.3.3      | InA UI Services .....                                | 228        |
| 6.3.4      | SQL Web API Services .....                           | 230        |
| <b>6.4</b> | <b>Testing Business Services .....</b>               | <b>230</b> |
| 6.4.1      | Use OData Service URLs .....                         | 231        |
| 6.4.2      | Use UI Previews .....                                | 231        |
| <b>6.5</b> | <b>Summary .....</b>                                 | <b>233</b> |
| <br>       |                                                      |            |
| <b>7</b>   | <b>Native SAP HANA Functions in CDS .....</b>        | <b>235</b> |
| <hr/>      |                                                      |            |
| <b>7.1</b> | <b>Implementation of a CDS Table Function .....</b>  | <b>236</b> |
| <b>7.2</b> | <b>Application Scenarios .....</b>                   | <b>244</b> |
| <b>7.3</b> | <b>Improve Performance and Avoid Errors .....</b>    | <b>245</b> |
| <b>7.4</b> | <b>Summary .....</b>                                 | <b>247</b> |
| <br>       |                                                      |            |
| <b>8</b>   | <b>Modeling Application Data .....</b>               | <b>249</b> |
| <hr/>      |                                                      |            |
| <b>8.1</b> | <b>Application Architecture in SAP S/4HANA .....</b> | <b>250</b> |
| <b>8.2</b> | <b>Field Labels .....</b>                            | <b>253</b> |
| 8.2.1      | Determination of a Field Label .....                 | 254        |
| 8.2.2      | Length of a Field Label .....                        | 255        |
| <b>8.3</b> | <b>Field Semantics .....</b>                         | <b>257</b> |
| 8.3.1      | Quantities and Amounts .....                         | 257        |
| 8.3.2      | Aggregation Behavior .....                           | 258        |
| 8.3.3      | System Times .....                                   | 260        |
| 8.3.4      | Text and Languages .....                             | 261        |
| 8.3.5      | Information for the Fiscal Year .....                | 262        |
| <b>8.4</b> | <b>Foreign Key Relations .....</b>                   | <b>262</b> |
| <b>8.5</b> | <b>Text Relations .....</b>                          | <b>267</b> |
| <b>8.6</b> | <b>Composition Relations .....</b>                   | <b>270</b> |
| <b>8.7</b> | <b>Time-Dependent Data .....</b>                     | <b>272</b> |
| <b>8.8</b> | <b>Summary .....</b>                                 | <b>274</b> |

|             |                                                                   |            |
|-------------|-------------------------------------------------------------------|------------|
| <b>9</b>    | <b>The Virtual Data Model of SAP S/4HANA .....</b>                | <b>275</b> |
| <hr/>       |                                                                   |            |
| <b>9.1</b>  | <b>Why a Virtual Data Model? .....</b>                            | <b>276</b> |
| <b>9.2</b>  | <b>SAP Object Types and Object Node Types .....</b>               | <b>277</b> |
| <b>9.3</b>  | <b>Categories of CDS Entities in the Virtual Data Model .....</b> | <b>280</b> |
| 9.3.1       | Basic Interface Views .....                                       | 280        |
| 9.3.2       | Composite Interface Views .....                                   | 281        |
| 9.3.3       | Consumption Views .....                                           | 283        |
| 9.3.4       | Other Types of VDMs .....                                         | 284        |
| <b>9.4</b>  | <b>Naming in the Virtual Data Model .....</b>                     | <b>285</b> |
| <b>9.5</b>  | <b>Basic Interface View for the Sales Order .....</b>             | <b>288</b> |
| 9.5.1       | View Annotations .....                                            | 288        |
| 9.5.2       | Structure of the Sales Order View .....                           | 292        |
| 9.5.3       | Specialization .....                                              | 293        |
| 9.5.4       | Element Annotations .....                                         | 294        |
| <b>9.6</b>  | <b>Tips for Finding Virtual Data Model Views .....</b>            | <b>295</b> |
| 9.6.1       | SAP Business Accelerator Hub and View Browser App .....           | 295        |
| 9.6.2       | Search in ABAP Development Tools .....                            | 298        |
| 9.6.3       | Search Views with Specific Annotations .....                      | 300        |
| 9.6.4       | ABAP Where-Used List .....                                        | 301        |
| <b>9.7</b>  | <b>Summary .....</b>                                              | <b>301</b> |
| <br>        |                                                                   |            |
| <b>10</b>   | <b>Modeling Analytical Applications .....</b>                     | <b>303</b> |
| <hr/>       |                                                                   |            |
| <b>10.1</b> | <b>Analytics in SAP S/4HANA .....</b>                             | <b>303</b> |
| <b>10.2</b> | <b>Analytical Views .....</b>                                     | <b>305</b> |
| 10.2.1      | First Analytical Cube View .....                                  | 305        |
| 10.2.2      | Test Environment for Analytical Views .....                       | 307        |
| 10.2.3      | Analytical Cube Views .....                                       | 311        |
| 10.2.4      | Analytical Dimension Views .....                                  | 314        |
| 10.2.5      | Analytical Model in the Test Environment .....                    | 320        |
| 10.2.6      | Consistency of the Analytical Model .....                         | 322        |
| <b>10.3</b> | <b>Analytical Queries .....</b>                                   | <b>325</b> |
| 10.3.1      | Definition of an Analytical Query .....                           | 325        |
| 10.3.2      | Initial Layout of a Query .....                                   | 329        |
| 10.3.3      | Filter, Select Options, Parameters .....                          | 332        |
| 10.3.4      | Calculation of Measures .....                                     | 338        |



|             |                                                          |            |
|-------------|----------------------------------------------------------|------------|
| 10.3.5      | Restricted Measures .....                                | 342        |
| 10.3.6      | Exception Aggregation .....                              | 344        |
| 10.3.7      | Currencies and Conversion .....                          | 351        |
| 10.3.8      | Analytical Query Selecting from Dimension Views .....    | 355        |
| <b>10.4</b> | <b>Analytical Infrastructure .....</b>                   | <b>356</b> |
| <b>10.5</b> | <b>Summary .....</b>                                     | <b>359</b> |
| <br>        |                                                          |            |
| <b>11</b>   | <b>Modeling Transactional Applications .....</b>         | <b>361</b> |
| <hr/>       |                                                          |            |
| <b>11.1</b> | <b>Transactional Applications .....</b>                  | <b>362</b> |
| <b>11.2</b> | <b>Transactional Infrastructure in SAP S/4HANA .....</b> | <b>363</b> |
| <b>11.3</b> | <b>Transactional Object Models .....</b>                 | <b>366</b> |
| 11.3.1      | Object Models .....                                      | 366        |
| 11.3.2      | Access Controls .....                                    | 370        |
| <b>11.4</b> | <b>Behavior Definitions .....</b>                        | <b>372</b> |
| 11.4.1      | Create Behavior Definition .....                         | 372        |
| 11.4.2      | Behavior Pool and Handler Implementation .....           | 381        |
| 11.4.3      | Consumption via EML .....                                | 385        |
| 11.4.4      | Static Field Control .....                               | 386        |
| 11.4.5      | Numbering .....                                          | 387        |
| 11.4.6      | Exclusive Locks .....                                    | 393        |
| 11.4.7      | Authorization Checks .....                               | 397        |
| 11.4.8      | Authorization Contexts and Privileged Access .....       | 401        |
| 11.4.9      | Associations .....                                       | 403        |
| 11.4.10     | Actions .....                                            | 407        |
| 11.4.11     | Functions .....                                          | 421        |
| 11.4.12     | Data Determinations and Validations .....                | 427        |
| 11.4.13     | Dynamic Feature Control .....                            | 437        |
| 11.4.14     | Mappings .....                                           | 442        |
| 11.4.15     | Calculated Fields .....                                  | 444        |
| 11.4.16     | Prechecks .....                                          | 445        |
| 11.4.17     | HTTP ETags .....                                         | 447        |
| 11.4.18     | Draft .....                                              | 449        |
| 11.4.19     | Side Effects .....                                       | 458        |
| 11.4.20     | Change Documents .....                                   | 461        |
| 11.4.21     | Events .....                                             | 464        |
| <b>11.5</b> | <b>Transactional Projection Object Models .....</b>      | <b>467</b> |
| 11.5.1      | Projection Object Models .....                           | 467        |
| 11.5.2      | Access Control .....                                     | 471        |

|              |                                                        |            |
|--------------|--------------------------------------------------------|------------|
| 11.5.3       | Denormalized Fields .....                              | 471        |
| 11.5.4       | Localized Elements .....                               | 472        |
| 11.5.5       | Calculated Fields/Virtual Elements .....               | 473        |
| <b>11.6</b>  | <b>Define Interface Behavior Definition .....</b>      | <b>477</b> |
| 11.6.1       | Create Interface Behavior Definition .....             | 477        |
| 11.6.2       | Static Feature Control .....                           | 479        |
| 11.6.3       | Operations .....                                       | 480        |
| 11.6.4       | Draft, ETag, and Side Effects .....                    | 481        |
| 11.6.5       | Events .....                                           | 482        |
| 11.6.6       | Consumption via EML .....                              | 483        |
| 11.6.7       | Release for Consumption .....                          | 483        |
| <b>11.7</b>  | <b>Define Projection Behavior Definition .....</b>     | <b>484</b> |
| 11.7.1       | Create Projection Behavior Definition .....            | 484        |
| 11.7.2       | Actions and Functions .....                            | 485        |
| 11.7.3       | Prechecks .....                                        | 487        |
| 11.7.4       | Augmentation .....                                     | 488        |
| 11.7.5       | Side Effects .....                                     | 491        |
| 11.7.6       | Events .....                                           | 492        |
| 11.7.7       | Consumption via EML .....                              | 492        |
| <b>11.8</b>  | <b>Runtime Orchestration .....</b>                     | <b>492</b> |
| 11.8.1       | Interaction Phase Operation Flow .....                 | 493        |
| 11.8.2       | Save Phase Operation Flow .....                        | 494        |
| 11.8.3       | Runtime Component Overview .....                       | 496        |
| 11.8.4       | Consumption via OData .....                            | 497        |
| <b>11.9</b>  | <b>SAP Fiori and OData Consumption .....</b>           | <b>498</b> |
| 11.9.1       | OData Service for Web API Consumption .....            | 498        |
| 11.9.2       | OData Service for UI Consumption .....                 | 499        |
| <b>11.10</b> | <b>SAP Event Mesh and Local Event Handlers .....</b>   | <b>510</b> |
| 11.10.1      | Local Event Handler .....                              | 510        |
| 11.10.2      | SAP Event Mesh .....                                   | 512        |
| <b>11.11</b> | <b>Summary .....</b>                                   | <b>513</b> |
| <br>         |                                                        |            |
| <b>12</b>    | <b>Hierarchies in CDS .....</b>                        | <b>515</b> |
| <hr/>        |                                                        |            |
| <b>12.1</b>  | <b>Hierarchy Categories and Basics .....</b>           | <b>516</b> |
| <b>12.2</b>  | <b>Annotation-Based Parent-Child Hierarchies .....</b> | <b>517</b> |
| 12.2.1       | Example of a Parent-Child Hierarchy .....              | 519        |
| 12.2.2       | Determination of a Hierarchy .....                     | 522        |

|             |                                                               |            |
|-------------|---------------------------------------------------------------|------------|
| 12.2.3      | Test a Hierarchy .....                                        | 524        |
| <b>12.3</b> | <b>CDS Hierarchies .....</b>                                  | <b>526</b> |
| 12.3.1      | Data for an Example of a Reporting Line Hierarchy .....       | 526        |
| 12.3.2      | Define the CDS Hierarchy .....                                | 529        |
| 12.3.3      | Hierarchy Attributes .....                                    | 531        |
| 12.3.4      | Visualization of a Hierarchy .....                            | 533        |
| 12.3.5      | Hierarchy with an Orphaned Node .....                         | 536        |
| 12.3.6      | Hierarchy with Multiple Parent Nodes .....                    | 537        |
| 12.3.7      | Hierarchy with Cycles .....                                   | 539        |
| 12.3.8      | Further Options for Defining Hierarchies .....                | 540        |
| 12.3.9      | CDS Hierarchies in ABAP SQL .....                             | 541        |
| 12.3.10     | OData Service for CDS Hierarchies .....                       | 543        |
| <b>12.4</b> | <b>Summary .....</b>                                          | <b>548</b> |
| <br>        |                                                               |            |
| <b>13</b>   | <b>CDS-Based Search Functionality .....</b>                   | <b>549</b> |
| <hr/>       |                                                               |            |
| <b>13.1</b> | <b>Modeling Value Helps .....</b>                             | <b>550</b> |
| 13.1.1      | Modeling Elementary Value Helps .....                         | 550        |
| 13.1.2      | Integrating Value Help CDS Views .....                        | 553        |
| 13.1.3      | Collective Value Helps .....                                  | 556        |
| 13.1.4      | Exposing Value Helps in OData Services .....                  | 558        |
| 13.1.5      | Using Value Helps .....                                       | 559        |
| <b>13.2</b> | <b>Free-Text Search Functionality in OData Services .....</b> | <b>570</b> |
| <b>13.3</b> | <b>Enterprise Search Functionality .....</b>                  | <b>578</b> |
| 13.3.1      | Define Enterprise Search Models .....                         | 578        |
| 13.3.2      | Adapt Enterprise Search Models from SAP .....                 | 580        |
| <b>13.4</b> | <b>Summary .....</b>                                          | <b>583</b> |
| <br>        |                                                               |            |
| <b>14</b>   | <b>Lifecycle and Stability .....</b>                          | <b>585</b> |
| <hr/>       |                                                               |            |
| <b>14.1</b> | <b>Stability Contracts .....</b>                              | <b>586</b> |
| <b>14.2</b> | <b>Lifecycle of Development Objects .....</b>                 | <b>590</b> |
| <b>14.3</b> | <b>Deprecation of Development Objects .....</b>               | <b>592</b> |
| <b>14.4</b> | <b>Use of CDS Models and Supported Capabilities .....</b>     | <b>594</b> |
| <b>14.5</b> | <b>Summary .....</b>                                          | <b>598</b> |

|             |                                                           |            |
|-------------|-----------------------------------------------------------|------------|
| <b>15</b>   | <b>Extensions of CDS Views and Other Entities .....</b>   | <b>599</b> |
| <hr/>       |                                                           |            |
| <b>15.1</b> | <b>Solution Variants and ABAP Language Versions .....</b> | <b>600</b> |
| <b>15.2</b> | <b>Stable CDS Extensions .....</b>                        | <b>602</b> |
| 15.2.1      | Stable Extensions of CDS Views .....                      | 603        |
| 15.2.2      | Example: Stable Extension Points .....                    | 604        |
| 15.2.3      | Example: Extension with Custom Fields .....               | 609        |
| 15.2.4      | CDS Extensions in Product Variants .....                  | 613        |
| <b>15.3</b> | <b>Extensions of Transactional Models .....</b>           | <b>615</b> |
| 15.3.1      | Add Fields to an Entity .....                             | 615        |
| 15.3.2      | Add Application Logic .....                               | 618        |
| 15.3.3      | Extend Action and Function Parameters and Results .....   | 620        |
| 15.3.4      | Extend Behavior .....                                     | 620        |
| 15.3.5      | Add Composition Child Entity .....                        | 622        |
| <b>15.4</b> | <b>Summary .....</b>                                      | <b>628</b> |
| <br>        |                                                           |            |
| <b>16</b>   | <b>Automated Testing .....</b>                            | <b>631</b> |
| <hr/>       |                                                           |            |
| <b>16.1</b> | <b>Test Logic of Data Selections .....</b>                | <b>631</b> |
| 16.1.1      | Fundamentals of the Test Double Framework .....           | 632        |
| 16.1.2      | Test Sample Overview .....                                | 633        |
| 16.1.3      | Test Implementations of CDS Views .....                   | 636        |
| 16.1.4      | Test ABAP Logic with SQL Accesses to CDS Views .....      | 650        |
| 16.1.5      | Test Code Generation Functions .....                      | 653        |
| <b>16.2</b> | <b>Test Logic of Transactional Applications .....</b>     | <b>658</b> |
| 16.2.1      | Test Behavior Handler .....                               | 658        |
| 16.2.2      | Test Events and Event Payloads .....                      | 661        |
| 16.2.3      | Test Event Handler .....                                  | 667        |
| 16.2.4      | Tests via EML Interface .....                             | 669        |
| <b>16.3</b> | <b>Summary .....</b>                                      | <b>670</b> |
| <br>        |                                                           |            |
| <b>17</b>   | <b>Troubleshooting .....</b>                              | <b>671</b> |
| <hr/>       |                                                           |            |
| <b>17.1</b> | <b>Performance Aspects .....</b>                          | <b>671</b> |
| 17.1.1      | Static View Complexity .....                              | 672        |

|             |                                                                              |            |
|-------------|------------------------------------------------------------------------------|------------|
| 17.1.2      | Calculated Fields .....                                                      | 675        |
| 17.1.3      | CDS Models in ABAP Code .....                                                | 678        |
| 17.1.4      | Performance Tests .....                                                      | 678        |
| 17.1.5      | Analysis Tools .....                                                         | 679        |
| <b>17.2</b> | <b>Pitfalls</b> .....                                                        | <b>687</b> |
| 17.2.1      | Null Values .....                                                            | 688        |
| 17.2.2      | Data Types .....                                                             | 691        |
| 17.2.3      | Decimal Shift Logic for Amounts .....                                        | 696        |
| <b>17.3</b> | <b>Troubleshoot Implementations of CDS Models</b> .....                      | <b>698</b> |
| 17.3.1      | Syntax Checks .....                                                          | 698        |
| 17.3.2      | Consistency Checks of Frameworks .....                                       | 702        |
| <b>17.4</b> | <b>Troubleshoot Activation Issues</b> .....                                  | <b>706</b> |
| 17.4.1      | Online Activation .....                                                      | 707        |
| 17.4.2      | Mass Checks and Repairs .....                                                | 708        |
| <b>17.5</b> | <b>Examine ABAP RESTful Application Programming Model Applications</b> ..... | <b>711</b> |
| <b>17.6</b> | <b>Summary</b> .....                                                         | <b>714</b> |

## **Appendices** 715

---

|          |                                                                          |            |
|----------|--------------------------------------------------------------------------|------------|
| <b>A</b> | <b>CDS Annotation Reference</b> .....                                    | <b>717</b> |
| <b>B</b> | <b>Migration to the ABAP RESTful Application Programming Model</b> ..... | <b>729</b> |
| <b>C</b> | <b>The Authors</b> .....                                                 | <b>737</b> |

|                     |     |
|---------------------|-----|
| Index .....         | 739 |
| Service Pages ..... | I   |
| Legal Notes .....   | II  |

## Index

- \\!@testing ..... 639, 657
- ?= operator ..... 193
- @AbapCatalog.compiler.
  - compareFilter ..... 717
- @AbapCatalog.enhancement .... 605, 717
- @AbapCatalog.entityBuffer.
  - definitionAllowed ..... 104, 717
- @AbapCatalog.extensibility ..... 607, 717
- @AbapCatalog.extensibility.
  - datasources ..... 608
- @AbapCatalog.extensibility.
  - extensible ..... 221
- @AbapCatalog.preserveKey ..... 717
- @AbapCatalog.
  - sqlViewAppendName ..... 717
- @AbapCatalog.sqlViewName ..... 717
- @AbapCatalog.typeSpec.
  - conversionExit ..... 145, 717
- @AccessControl.auditing.
  - specification ..... 191
- @AccessControl.auditing.type ..... 191
- @AccessControl.authorization-
  - Check ..... 60, 170, 200, 702-704, 718
- @AccessControl.privileged-
  - Associations ..... 205, 718
- @Aggregation ..... 259
- @Aggregation.default ..... 311, 718
- @Aggregation.default\
  - #FORMULA ..... 342
- @Analytics.dataCategory ..... 195, 311, 314, 320, 718
  - #CUBE ..... 306
- @Analytics.internalName ..... 295, 359, 719
- @Analytics.query ..... 194, 549, 704, 705, 719
- @Analytics.technicalName ..... 307, 327, 328, 358, 719
- @AnalyticsDetails.
  - exceptionAggregationSteps ... 346, 719
- @AnalyticsDetails.query.axis .... 331, 719
- @AnalyticsDetails.query.
  - display ..... 331, 719
- @AnalyticsDetails.query.
  - formula ..... 340, 719
- @AnalyticsDetails.query.hidden ..... 719
- @AnalyticsDetails.query.
  - sortDirection ..... 331, 719
- @AnalyticsDetails.query.
  - totals ..... 331, 719
- @API.element.releaseState ..... 593, 719
- @API.element.successor ..... 594, 719
- @ClientHandling.algorithm ..... 720
- @ClientHandling.type ..... 720
- @CompatibilityContract ..... 587
- @Consumption.dbHints ..... 675, 720
- @Consumption.defaultValue .... 336, 720
- @Consumption.derivation ..... 337, 596, 720
  - @Consumption.derivation.
    - lookupEntity ..... 195
- @Consumption.filter ..... 336, 549, 720
- @Consumption.hidden ..... 144, 335, 575, 720
  - @Consumption.ranked ..... 577
- @Consumption.valueHelp ..... 552
  - @Consumption.valueHelpDefault.
    - binding.usage ..... 553, 566
- @Consumption.valueHelpDefault.
  - display ..... 720
- @Consumption.valueHelpDefault.
  - display ..... 553, 563, 566
- @Consumption.valueHelp-
  - Definition ..... 195, 216, 336, 549, 552, 556
- @Consumption.valueHelpDefinition.
  - additionalBinding ..... 555, 557
- @Consumption.valueHelpDefinition.
  - additionalBinding.element ..... 721
- @Consumption.valueHelpDefinition.
  - additionalBinding.localElement .... 721
- @Consumption.valueHelpDefinition.
  - additionalBinding.usage ..... 564
- @Consumption.valueHelpDefinition.
  - association ..... 555, 720
- @Consumption.valueHelpDefinition.
  - entity ..... 556
- @Consumption.valueHelpDefinition.
  - entity.element ..... 555, 558, 721
- @Consumption.valueHelpDefinition.
  - entity.name ..... 555, 721
- @Consumption valueHelpDefault.
  - binding ..... 720
- @DefaultAggregation ..... 259, 311, 356, 721
- @EndUserText ..... 51, 145

@EndUserText.heading ..... 721  
 @EndUserText.label ... 34, 60, 67, 76, 149, 151, 157, 232, 253, 254, 699, 721  
 @EndUserText.quickInfo ..... 255, 721  
 @EnterpriseSearch.enabled ..... 579, 581, 721  
 @EnterpriseSearch.freeStyleField ..... 579, 580, 721  
 @EnterpriseSearch.model ..... 578, 580, 721  
 @EnterpriseSearch.modelName ..... 722  
 @EnterpriseSearch.  
   modelNamePlural ..... 722  
 @EnterpriseSearch.navigation ..... 722  
 @EnterpriseSearch.  
   responseField ..... 579, 722  
 @EnterpriseSearch.  
   responseItemKey ..... 722  
 @EnterpriseSearch.  
   resultItemKey ..... 579, 580  
 @EnterpriseSearch.title ..... 722  
 @Environment.systemField ..... 95, 144, 335, 336, 722  
 @Hierarchy.parentChild ..... 518, 722  
 @Hierarchy.parentChild.directory ... 196  
 @MappingRole ..... 166, 722  
 @Metadata.allowExtensions ..... 156, 157, 722  
 @Metadata.  
   ignorePropagatedAnnotations ..... 73, 148, 152, 155, 159, 722  
 @Metadata.layer ..... 156, 157, 723  
 @MetadataExtension.  
   usageAllowed ..... 158, 723  
 @ObjectModel.alternativeKey ... 138, 139  
 @ObjectModel.alternativeKey.  
   element ..... 139, 723  
 @ObjectModel.alternativeKey.  
   id ..... 139, 723  
 @ObjectModel.alternativeKey.  
   uniqueness ..... 723  
 @ObjectModel.association.type ..... 270  
 @ObjectModel.collectiveValueHelp.  
   for.element ..... 556  
 @ObjectModel.compositionRoot ..... 271  
 @ObjectModel.dataCategory .... 140, 196, 316, 551, 723  
   #HIERARCHY ..... 518  
   #TEXT ..... 269  
 @ObjectModel.filter.enabled ..... 723  
 @ObjectModel.filter.  
   transformedBy ..... 723  
 @ObjectModel.foreignKey.  
   association ..... 264, 549, 723  
 @ObjectModel.hierarchy.  
   association ..... 518, 723  
 @ObjectModel.lifecycle.enqueue.  
   expirationInterval ..... 723  
 @ObjectModel.modelingPattern ..... 102, 103, 595, 723  
 @ObjectModel.representativeKey ... 266, 269, 314, 316, 551, 724  
 @ObjectModel.sapObjectType.  
   name ..... 724  
 @ObjectModel.  
   sapObjectTypeReference ..... 146, 280, 724  
 @ObjectModel.semanticKey ..... 581  
 @ObjectModel.sort.enabled ..... 724  
 @ObjectModel.sort.transformedBy ... 724  
 @ObjectModel.supported-  
   Capabilities ..... 291, 551, 594, 673, 724  
 @ObjectModel.text ..... 315  
 @ObjectModel.text.association ..... 267, 316, 724  
 @ObjectModel.text.element ..... 267, 318, 724  
 @ObjectModel.usageType ..... 291  
 @ObjectModel.usageType.  
   dataClass ..... 145, 291, 724  
 @ObjectModel.usageType.  
   serviceQuality ..... 144, 291, 724  
 @ObjectModel.usageType.  
   serviceQuality... ..... 672  
 @ObjectModel.usageType.  
   sizeCategory ..... 144, 291, 724  
 @ObjectModel.virtualElement ..... 724  
 @ObjectModel.  
   virtualElementCalculatedBy ..... 473, 474, 724  
 @OData.entityType.name ..... 143, 220, 221, 724  
 @OData.publish ..... 212, 724  
 @Scope ..... 134, 724  
 @Search ..... 570  
 @Search.defaultSearchElement ..... 571, 581, 583, 725  
 @Search.fuzzinessThreshold ..... 571, 574, 725  
 @Search.ranking ..... 573, 574, 577, 725  
 @Search.searchable ..... 571, 725  
 @Semantics.amount ..... 257

@Semantics.amount.  
   currencyCode ..... 96, 97, 101, 339, 697, 725  
 @Semantics.booleanIndicator .... 693, 725  
 @Semantics.businessDate.from 273, 725  
 @Semantics.businessDate.to .... 273, 725  
 @Semantics.currencyCode ..... 257, 725  
 @Semantics.dateTime ..... 725  
 @Semantics.fiscal ..... 262  
 @Semantics.fiscal.dayOfYear ..... 725  
 @Semantics.fiscal.period ..... 725  
 @Semantics.fiscal.quarter ..... 725  
 @Semantics.fiscal.week ..... 725  
 @Semantics.fiscal.year ..... 726  
 @Semantics.fiscal.yearPeriod ..... 726  
 @Semantics.fiscal.yearQuarter ..... 726  
 @Semantics.fiscal.yearVariant ..... 726  
 @Semantics.fiscal.yearWeek ..... 726  
 @Semantics.language ..... 67, 261, 269  
 @Semantics.quantity ..... 257  
 @Semantics.quantity.  
   unitOfMeasure ... 96, 97, 100, 134, 135, 142, 149, 151, 154, 342  
 @Semantics.systemDate ..... 260  
 @Semantics.systemDate.  
   createdAt ..... 134, 135, 139, 141, 142, 149, 151, 726  
 @Semantics.systemDate.  
   lastChangedAt ..... 726  
 @Semantics.systemDateTime ..... 260  
 @Semantics.systemDateTime.  
   createdAt ..... 726  
 @Semantics.systemDateTime.  
   lastChangedAt ..... 726  
 @Semantics.systemTime ..... 260  
 @Semantics.systemTime.  
   createdAt ..... 726  
   lastChangedAt ..... 726  
 @Semantics.text ..... 261, 726  
 @Semantics.unitOfMeasure ..... 257, 726  
 @Semantics.user.createdBy ..... 726  
 @Semantics.user.lastChangedBy ..... 726  
 @Semantics.uuid ..... 693, 727  
 @UI.facet ..... 727  
 @UI.fieldGroup ..... 727  
 @UI.headerInfo ..... 727  
 @UI.hidden ..... 727  
 @UI.identification ..... 500, 727  
 @UI.lineItem ..... 232, 500, 556, 727  
 @UI.lineItem.importance ..... 157, 727  
 @UI.selectionField ..... 232, 500, 556, 727

@UI.textArrangement ..... 331, 727  
 @VDM.lifecycle.contract.type ... 284, 285, 591, 727  
 @VDM.lifecycle.status ..... 593, 594, 728  
 @VDM.lifecycle.successor ..... 593, 594, 728  
 @VDM.private ..... 284, 728  
 @VDM.private\  
   true ..... 591  
 @VDM.usageType ..... 728  
 @VDM.viewExtension ..... 728  
 @VDM.viewExtension\  
   true ..... 613  
 @VDM.viewType ..... 280, 281, 283, 728  
 @VDM.viewType\  
   #EXTENSION ..... 607  
 #AVG ..... 259, 344  
 #COUNT ..... 344  
 #DEPRECATED ..... 593, 594  
 #FIRST ..... 344  
 #LAST ..... 345  
 #MAX ..... 311, 344  
 #MIN ..... 311, 344  
 #NONE ..... 311  
 #NOP ..... 311  
 #PUBLIC\_LOCAL\_API ..... 591  
 #SAP\_INTERNAL\_API ..... 591  
 #STD ..... 344  
 #SUM ..... 311, 344  
 \$node ..... 531  
 \$parameters ..... 92  
 \$projection ..... 89  
 \$session.client ..... 70, 71, 238  
 \$session.system\_date ..... 70, 94  
 \$session.system\_language ..... 70  
 \$session.user ..... 70, 189  
 \$session.user\_date ..... 70  
 \$session.user\_timezone ..... 70

## A

A2A communication ..... 361  
 ABAP application  
   infrastructure ..... 253, 277  
 ABAP application server ..... 364, 365  
 ABAP Cloud ..... 601, 613, 729  
 ABAP Cloud Development ..... 617  
 ABAP cross trace ..... 671, 711  
 ABAP Data Dictionary ..... 262, 409, 410  
   domain ..... 145  
   structure ..... 409

ABAP Development Tools (ADT) ... 19, 23,  
   28, 31, 162, 365, 632, 698  
   *editor* ..... 132  
   *environment* ..... 653  
   *test classes* ..... 653  
 ABAP documentation ..... 50  
 ABAP for Cloud Development ..... 601  
 ABAP host variable ..... 45  
 ABAP in Eclipse ..... 19, 365  
 ABAP infrastructure ... 396, 401, 408, 428,  
   432, 437, 443, 452, 466, 491, 499  
 ABAP language ..... 365, 510  
 ABAP language version ..... 617  
   2 ..... 601  
   5 ..... 601  
 ABAP-Managed Database Procedures  
   (AMDP) ..... 235  
   *class method* ..... 238  
   *procedure* ..... 239  
 ABAP platform ..... 17, 365, 461, 472, 492  
 ABAP programming model  
   for SAP Fiori ..... 364, 729  
 ABAP RESTful application programming  
   model ..... 17, 57, 102, 211, 361, 364,  
   385, 447, 662, 671, 711, 729  
   *runtime component* ..... 497  
 ABAP session ..... 450  
 ABAP SQL ... 23, 24, 36, 44, 58, 71, 95, 365,  
   397, 650  
   *interface* ... 144, 166, 194, 633, 642, 678  
   *new syntax* ..... 45  
   *test double framework* ..... 633, 650  
   *trace* ..... 680, 686  
 ABAP system field ..... 95  
 ABAP unit test ..... 633, 638  
 ABAP Workbench ..... 28, 30, 31  
 Access condition ..... 163, 173, 186  
   *default true* ..... 183  
   *literal values* ..... 192  
 Access control ..... 71, 113, 161, 172, 290,  
   370, 397, 471  
   *analytical queries* ..... 194  
   *aspect* ..... 190  
   *block standard data selection* ..... 203  
   *creation template* ..... 162  
   *decoupling from input* ..... 205  
   *field level* ..... 199  
   *fields* ..... 199  
   *implementation* ..... 171  
   *inheritance* ..... 180, 182  
   *language* ..... 364  
   *mode of action* ..... 166  
   Access control (Cont.)  
     *path expression* ..... 171  
     *redefine* ..... 200  
     *SAP-delivered models* ..... 200  
     *testing* ..... 207  
     *unique key* ..... 168  
     *without authorization objects* ..... 189  
   Access rule ..... 163  
   Action ... 361, 363, 372, 397, 407, 491, 620  
     *parameters* ..... 409  
   Activation log ..... 701  
   Activation problem ..... 40, 111  
   Active and inactive CDS models ..... 40  
   Aggregation ..... 258  
     *hierarchical* ..... 543  
     *level* ..... 87  
     *type* ..... 259  
   All quantifier ..... 177  
   ALV Tree Control ..... 533  
   Amount calculation ..... 339  
   Amount field ..... 257, 287  
   ANALYTICAL\_CUBE ..... 596  
   ANALYTICAL\_DIMENSION ..... 596  
   ANALYTICAL\_PARENT\_CHILD\_  
     HIERARCHY\_NODE ..... 597  
   ANALYTICAL\_PROVIDER ..... 596  
   ANALYTICAL\_QUERY ..... 103, 194, 215,  
     230, 549, 596  
   Analytical application ..... 303  
   Analytical cube ..... 282  
   Analytical engine ..... 56, 101  
   Analytical formula ..... 342  
   Analytical measure ..... 259  
   Analytical dimension view ..... 314  
   Analytical engine ..... 194, 253, 357, 515  
   Analytical interface ..... 358  
   Analytical model ..... 320, 322  
   Analytical processor ..... 357, 515  
   Analytical query ..... 56, 107, 194, 215,  
     228, 325  
     *calculate measures* ..... 338  
     *define* ..... 325  
     *define variables* ..... 332  
     *exception aggregation* ..... 344  
     *extension* ..... 608  
     *layout* ..... 329  
     *restricted measures* ..... 342  
     *select from dimension views* ..... 355  
   Analytics ..... 303  
     *data category* ..... 311  
     *modeling* ..... 303  
     *tools* ..... 358

Annotated formula ..... 341  
 Annotate view ..... 55, 143, 156  
 Annotation ..... 54, 131, 463, 499  
   *active* ..... 50, 132, 149, 151, 155,  
     157, 158  
   *aggregation behavior* ..... 258  
   *array* ..... 138, 148  
   *array of* ..... 138  
   *artifacts* ..... 143  
   *authorization* ..... 170  
   *composition relations* ..... 270  
   *consistency aspects* ..... 153  
   *default value* ..... 141  
   *definition* ..... 132, 587  
   *document* ..... 144  
   *domain* ..... 132, 134, 136, 158  
   *effect* ..... 143  
   *enum* ..... 140  
   *errors* ..... 705  
   *explicit* ..... 254  
   *field* ..... 257  
   *fiscal year* ..... 262  
   *foreign key* ..... 264  
   *fully qualified name* ..... 136  
   *grouping* ..... 137  
   *main* ..... 136  
   *name* ..... 136  
   *propagation* ..... 367  
   *propagation logic* ... 51, 140, 142, 147,  
     254, 312  
   *root* ..... 136  
   *runtime environment* ..... 144  
   *scope* ..... 142  
   *search* ..... 574  
   *stability* ..... 587  
   *subannotation* ..... 136  
   *system times* ..... 260  
   *text and language* ..... 261  
   *text relations* ..... 267  
   *type* ..... 139  
   *undefined* ..... 699  
   *value* ..... 136, 139  
 annotation  
   *definition* ..... 587  
 Annotation-based hierarchy ..... 517  
 API state ..... 588, 590  
 Application architecture ..... 249  
 Application data modeling ..... 249  
 Application infrastructure ..... 249  
 Application performance ..... 671  
   *testing* ..... 678  
 Application programming  
   interface (API) ..... 227, 276  
   *state* ..... 290  
 as ..... 35  
 AS PARENT CHILD HIERARCHY  
   keyword ..... 530  
 Aspect ..... 190  
   *self-defined* ..... 191  
 aspect pfcg\_auth ..... 207  
 aspect user ..... 189  
 Assert statement ..... 642  
 Association ..... 27, 54, 107, 367  
   *annotations* ..... 142  
   *cardinality* ..... 109  
   *composition* ..... 111  
   *default filter* ..... 126, 127  
   *define* ..... 77, 108  
   *exposure* ..... 111  
   *in ABAP code* ..... 129  
   *in CDS view* ..... 117  
   *name* ..... 108, 288  
   *projected* ..... 117  
   *redirected to composition child* ... 214  
   *redirected to parent* ..... 214  
   *remove* ..... 111  
   *stability* ..... 587  
   *target* ..... 110  
   *to parent* ..... 112, 113, 214  
 Audit ..... 191  
 Augmentation ..... 488  
 Authority check ..... 620  
 Authorization ..... 161, 361, 621, 624  
   *control* ..... 161, 362, 372  
   *database level* ..... 166  
   *extension fields* ..... 200  
   *field* ..... 163  
   *map fields* ..... 207  
   *master* ..... 624  
   *object* ..... 163, 397, 401  
   *protection* ..... 170  
   *role* ..... 167  
 Authorization check ..... 170, 195, 207,  
   397, 401  
   *change* ..... 203  
   *documentation* ..... 171  
 Authorization context  
   *own* ..... 621  
   *privileged* ..... 620  
 Automated testing ..... 207  
 Average calculation ..... 258  
 avg ..... 88

**B**

Basic interface views ..... 280, 288, 367  
*redundancies* ..... 281  
 Behavior definition ..... 361, 372, 658  
 Buffer  
   *layer core* ..... 105  
   *layer customer* ..... 106  
   *type none* ..... 106  
   *type single* ..... 105  
 Business add-in (BAI) ..... 618, 620, 621, 711  
 Business logic ..... 361–363, 365, 366, 372, 429, 467  
 Business object ..... 270, 362, 366, 372, 467, 492  
   *node* ..... 270  
 Business Object Processing Framework (BOPF) ..... 390, 428, 729  
 Business process ..... 362, 467  
 Business services  
   *benefits* ..... 211  
   *InA UI services* ..... 228  
   *OData UI services* ..... 222  
   *OData Web API service* ..... 227  
   *testing* ..... 231  
 Bypass when statement ..... 178

**C**

Calculated field  
   *value* ..... 675  
 Camel case notation ..... 35, 237  
 Cardinality ..... 25, 109, 114, 122, 124, 125, 179  
   *exact one* ..... 84  
   *foreign key association* ..... 264  
   *one to many* ..... 84  
   *to exact one* ..... 84  
   *to many* ..... 84  
   *to one* ..... 84  
 Case statement ..... 68, 691  
 Cast function ..... 62, 76, 148, 151, 254  
   *nest* ..... 63  
 Cast operation ..... 155, 159  
 CDS\_CLIENT ..... 71, 237, 289  
 CDS\_MODELING\_ASSOCIATION\_TARGET ..... 595  
 CDS\_MODELING\_DATA\_SOURCE ..... 595  
 CDS abstract entity ..... 409, 411, 464  
 CDS access control ..... 636, 678  
   *testing* ..... 643  
 CDS editor ..... 34, 36, 236  
   *auto-completion* ..... 40  
   *code completion* ..... 135, 700  
 CDS entity  
   *abstract entity* ..... 55  
   *custom entity* ..... 55  
   *hierarchy* ..... 55  
   *projection view* ..... 54  
   *semantic names* ..... 219  
   *specialized* ..... 219  
   *table function* ..... 55  
   *transient view* ..... 54  
   *type* ..... 53, 54  
   *view entity* ..... 54, 55  
 CDS extension ..... 602, 604, 612, 614  
 CDS extension include ..... 604, 606, 610, 614  
 CDS extension include view ..... 608  
 CDS hierarchy ..... 526  
   *entity* ..... 57  
 CDS metadata extension ..... 54, 55  
 CDS model ..... 23, 54, 249  
   ABAP ..... 678  
   *activate* ..... 35  
   *activation logic* ..... 708  
   *create* ..... 31  
   *denormalize* ..... 123  
   *error message* ..... 702  
   *inconsistencies* ..... 154  
   *inheritance* ..... 182  
   *key* ..... 168  
   *names* ..... 35, 61, 171  
   *parameters* ..... 92  
   *propagation logic* ..... 155  
   *readability* ..... 155  
   *reuse* ..... 672  
   *SAP-delivered* ..... 158  
   *SQL-friendly* ..... 672  
   *static complexity* ..... 672  
   *template* ..... 33  
   *transparency* ..... 62  
   *troubleshoot* ..... 698  
   *versions* ..... 39  
 CDS role ..... 162–164  
   *can't be fulfilled* ..... 203  
   *direct reference* ..... 189  
   *inheritance* ..... 184  
   *mapping role* ..... 166  
   *multiple roles* ..... 171  
   *optional elements* ..... 186  
   *path expressions* ..... 179  
   *redefinition* ..... 201

CDS simple type ..... 62  
 CDS table function .. 57, 72, 235, 363, 677  
   *associations and annotations* ..... 238  
   *client-dependent* ..... 237  
   *components* ..... 237  
   *implement* ..... 236  
   *test* ..... 241  
 CDS test double framework ..... 633, 639  
 CDS view ..... 23, 363  
   *activation* ..... 370  
   *aggregating* ..... 87  
   *client field* ..... 61  
   *create* ..... 31, 40, 45  
   *definition* ..... 41  
   *display with extensions* ..... 613  
   *edit* ..... 37  
   *explicit joins* ..... 122  
   *extension* ..... 246  
   *fields* ..... 61  
   *hierarchy* ..... 40  
   *internal logic* ..... 107  
   *model* ..... 60  
   *path expressions* ..... 120  
   *projection* ..... 212  
   *search* ..... 298, 300, 301  
   *stack* ..... 46, 47, 148, 159, 638  
   *static complexity* ..... 672  
   *syntax* ..... 58  
   *test* ..... 41  
   V1 ..... 55  
   V2 ..... 55  
   *with inner join* ..... 85  
 Change document object ..... 461  
 Change documents ..... 461, 624  
 Characteristic ..... 308  
 Check ..... 698  
 Check table ..... 263  
 Child node ..... 270  
 CHILD TO PARENT ASSOCIATION ..... 530  
 CL\_ABAP\_BEHV\_TEST\_ENVIRONMENT ..... 667  
 CL\_ABAP\_UNIT\_ASSERT ..... 642  
 CL\_CDS\_TEST\_DATA ..... 647  
 CL\_CDS\_TEST\_ENVIRONMENT ..... 639  
 CL\_RAP\_EVENT\_TEST\_ENVIRONMENT ..... 662  
 Class method ..... 238  
 Class setup method ..... 639  
 Class teardown method ..... 643  
 Client ..... 61, 70, 71  
   *field* ..... 72  
   *handling* ..... 71, 289  
 Client-dependent data ..... 289  
 client specified ..... 71  
 Cloud development ..... 592, 601  
 coalesce ..... 691  
 Code ..... 700  
   *completion* ..... 135, 700  
 collapsed ..... 547  
 Collective value help ..... 556  
 combination mode and statement .... 202  
 combination mode or statement ..... 201  
 Comment ..... 60  
 COMMIT ENTITIES ..... 663  
   *IN SIMULATION MODE* ..... 495  
 COMMIT WORK ..... 447, 495  
 Compatible change ..... 586  
 Complexity metrics ..... 48  
 Composite interface view ..... 281, 282  
 Composition ..... 623  
   *association* ..... 270  
   *hierarchy* ..... 112  
   *relationship* ..... 362  
   *restrictions* ..... 114  
 composition..of ..... 112  
 concat ..... 60  
 Consistency check ..... 306  
 Consistency condition ..... 324  
 Consumption view ..... 283  
   *compatibility* ..... 283  
   *not released* ..... 283  
 Contains function ..... 574  
 Content ID references ..... 418  
 Conversion exit ..... 62, 145, 694, 695  
 Conversion function ..... 89, 99, 636, 646  
   *analytical queries* ..... 101  
   *error handling* ..... 101  
   *parameters* ..... 100  
   *performance aspects* ..... 89  
 Core data services (CDS) ..... 17, 363  
   *abstract entity* ..... 57, 409  
   *associations* ..... 27  
   *custom entities* ..... 57  
   *data access in ABAP* ..... 44  
   *element* ..... 107  
   *embed conversion functions* ..... 99  
   *hierarchies* ..... 541  
   *implementation errors* ..... 687  
   *modeling* ..... 53  
   *projection view* ..... 56  
   *repair tool* ..... 708  
   *service-specific views* ..... 211  
   *session variable* ..... 69  
   *supported capabilities* ..... 594

Core data services (CDS) (Cont.)  
*syntax* ..... 58  
*type conversions* ..... 63  
*version 2* ..... 366  
 Cost center ..... 522  
*hierarchy* ..... 519  
 count(\*) ..... 60, 88  
 COUNT different values ..... 258  
 count distinct ..... 88, 259, 690  
 Counter ..... 287, 345, 355  
 Create, read, update, delete (CRUD) ... 729  
*operations* ..... 361, 363, 372, 376, 497  
 Create object ..... 667  
 Create statement ..... 90, 121, 124, 125  
 Creation dialog ..... 216, 222  
 Creation wizard ..... 28  
 Cross-client data access ..... 71  
 Cube ..... 312  
 Cube view ..... 303, 305, 311  
*analytic* ..... 306  
*define* ..... 305  
 cuky ..... 96  
 curr ..... 96, 97, 696  
*fields* ..... 97  
 curr\_to\_decfloat\_amount .... 97, 342, 355  
 Currency conversion .... 99, 100, 351, 353, 355, 635, 647  
 Currency field ..... 257  
 Currency shift ..... 97  
 Custom entities ..... 445  
 Customer namespace ..... 26  
 Custom field ..... 246  
 Cycle ..... 540

**D**

DATA\_STRUCTURE ..... 595  
 Database procedure ..... 240  
 Database table  
*create* ..... 28  
*editor* ..... 29  
*simplify* ..... 31  
 Database view ..... 36  
*implicit join* ..... 121  
 Data control language (DCL) ..... 162  
 Data definition language (DDL) ..... 53  
 Data definition language  
 source (DDLs) ..... 32, 33, 36, 54, 60  
 Data element ..... 62  
*field label* ..... 255  
*medium text* ..... 256

Data model  
*define* ..... 24  
*implement* ..... 26  
 Data preview ..... 42  
 Data record ..... 42  
*duplicate* ..... 110  
*missing* ..... 110  
 Data redundancy ..... 179  
 Data selection ..... 53  
*privileged* ..... 193, 204  
 DCLS ..... 162, 171, 191  
 dd\_cds\_calculated\_unit ..... 342  
 DDLA ..... 132  
 DDLX ..... 54, 57  
 Decimal shift ..... 97  
*logic* ..... 696  
 DECOMMISSIONED ..... 591  
 Decommissioning ..... 592, 594  
 Default filter ..... 126, 127  
 define abstract entity ..... 55  
 define accesspolicy ..... 191  
 define custom entity ..... 55  
 define hierarchy entity ..... 55  
 define statement ..... 59  
 define table function ..... 55  
 define type ..... 55  
 define view ..... 54  
*entity* ..... 34, 54, 55, 59  
*projection entity* ..... 54  
*transient entity* ..... 54  
 Delegation approach ..... 184  
 Denormalization ..... 81, 123  
 Denormalized fields ..... 488  
 Dependency Analyzer ..... 46  
 DEPRECATED ..... 591, 592  
 Deprecation ..... 585, 592, 594  
*period* ..... 591  
*propagation* ..... 593  
*SAP-internal* ..... 593  
*types* ..... 592  
 Derivation ..... 337  
 DERIVATION\_FUNCTION ..... 596  
 Descendants ..... 516, 533  
 Design phase ..... 24  
 Determination .... 383, 428, 432, 445, 452, 454, 618  
*execution times* ..... 429  
 Determine action ..... 429, 436, 452, 619  
 Developer extensibility ..... 601, 604, 610, 613  
 Dimension ..... 107, 308, 311  
*analytic* ..... 314

Dimension (Cont.)  
*consistency* ..... 322  
*field* ..... 311, 314  
*multiple key fields* ..... 324  
*replace* ..... 324  
*view* ..... 303, 355  
 Directory...filter by ..... 541  
 Display attribute ..... 331  
 Display authorization ..... 164  
 Display currency ..... 351  
 Distinct statement ..... 74  
 Draft ..... 361, 625  
 Draft concept ..... 449  
 Draft data ..... 603  
 Draft lifecycle handler ..... 453  
 Draft orchestration ..... 451, 453  
 Draft query view ..... 199  
 Draft table ..... 450  
 DrillState ..... 547  
 DRTY ..... 54  
 Duplicate record ..... 78  
 Duration ..... 287  
 Dynamic variable ..... 336

**E**

Element annotations ..... 311  
 Embedded analytics ..... 304  
 Enqueue function modules ..... 395  
 Enqueue object ..... 361, 437, 450  
 Enterprise search ..... 578–580  
*adapt models* ..... 580  
*model* ..... 550  
 Entity buffer definition ..... 55, 58, 104  
 Entity Data Model (EDM) ..... 691  
 Entity key ..... 273  
 Entity Manipulation  
 Language (EML) ..... 385, 401, 405, 410, 417, 437, 458, 480, 483, 488, 492, 493, 497, 615, 625, 667, 670, 729  
 Entity relationship model (ERM) ... 27, 41  
 Entity view ..... 263  
 Enumeration value ..... 140  
 ETag ..... 447, 453, 484, 624  
 Event ..... 361, 492, 510, 621, 661  
 Event binding ..... 512  
 Event handler ..... 661, 667  
 Event payload ..... 661, 663  
 Except element ..... 80  
 Exception aggregation ..... 344, 349, 355  
*aggregation behavior* ..... 344  
*steps* ..... 346  
 exceptionAggregationBehavior ..... 344  
 exceptionAggregationSteps ..... 347  
 Exchange rate type ..... 351  
 Exclusive lock ..... 362, 372  
 expanded ..... 547  
 extend abstract entity ..... 55  
 extend custom entity ..... 55  
 Extend view ..... 55, 57, 142, 200  
*entity* ..... 55, 57  
 Extension ..... 599  
*SAP standard fields* ..... 608  
 Extension association ..... 246, 604, 607, 608, 614  
 Extension category ..... 605  
 Extension field  
*suffix* ..... 605  
 Extension include ..... 604  
 Extension include view ..... 111, 203, 293  
 Extension object ..... 602  
 Extension of software ..... 602  
 Extension point ..... 602  
*persistence* ..... 603  
*stable* ..... 604, 614  
 EXTRACTION\_DATA\_SOURCE ..... 596

**F**

F2 help ..... 50, 64, 98  
 Factory action ..... 418  
 Factory calendar ..... 244  
 Fact view ..... 320  
 Feature control ..... 438, 621  
*dynamic* ..... 361, 385, 427, 437  
*static* ..... 385  
 Field  
*abbreviation* ..... 286  
*annotation* ..... 257, 294  
*control* ..... 620  
*denormalized* ..... 488  
*length extension* ..... 588  
*variable* ..... 336  
 Field label ..... 253  
*determine* ..... 254  
*length* ..... 255, 256  
 Filter criteria ..... 675, 677  
 Fiscal year ..... 262  
 Foreign key ..... 262, 263  
*table* ..... 263  
*view* ..... 263  
 Foreign key association ..... 110, 264, 266, 314, 317  
*define* ..... 265



- Foreign key association (Cont.)
    - use* ..... 267
  - Formula in query ..... 340
  - for testing ..... 640
  - Free-text search ..... 549, 570, 571
  - From statement ..... 123
  - Full access rule ..... 202
  - Function ..... 361, 363, 372, 397, 421, 620
    - currency\_conversion* ..... 100
    - fltp\_to\_dec* ..... 63–65
    - unit\_conversion* ..... 100
  - Fundamental data model ..... 24
  - Fuzziness ..... 571
  - Fuzzy search ..... 550
- ## G
- get\_numeric\_value* ..... 97
  - Globally unique identifier (GUID) ..... 390, 432
  - Group by statement ..... 59, 87–89
- ## H
- Handler ..... 658
  - Helper field ..... 575
  - Hierarchy ..... 515, 596, 597
    - ABAP SQL* ..... 541
    - aggregation* ..... 543
    - association* ..... 518
    - attributes* ..... 531, 532
    - base entity* ..... 518
    - cache* ..... 541
    - create* ..... 40
    - cycle* ..... 516, 533, 539
    - data source* ..... 530
    - define* ..... 529
    - depth* ..... 541
    - descendants* ..... 542
    - determine* ..... 522
    - directory* ..... 517, 521, 522, 524, 541
    - error source* ..... 523
    - generate spantree* ..... 541
    - load* ..... 541
    - mixed* ..... 523
    - multiple parent nodes* ..... 537
    - navigation functions* ..... 541, 542
    - node type* ..... 541
    - node view* ..... 518
    - OData services* ..... 543
    - ordinal number* ..... 533, 538
    - orphaned node* ..... 536

- Hierarchy (Cont.)
  - parameters* ..... 541
  - relational data model* ..... 517
  - strict* ..... 516
  - structure* ..... 518
  - syntax limitations* ..... 530
  - test* ..... 524
  - time-dependent* ..... 541
  - view* ..... 518
  - visualizations* ..... 533
- HIERARCHY\_DESCENDANTS ..... 542
- hierarchy\_is\_cycle* ..... 540
- hierarchy\_is\_orphan* ..... 533, 537
- hierarchy\_level* ..... 533
- hierarchy\_parent\_rank* ..... 533
- hierarchy\_rank* ..... 533, 538
- hierarchy\_tree\_size* ..... 533
- Hierarchy engine ..... 517, 531, 533, 541
  - cycle* ..... 540
- Hierarchy node ..... 516, 522
  - level* ..... 517
- HierarchyNodes ..... 546
- HierarchyQualifier ..... 546
- HTTP ..... 363, 383

## I

- I\_CalendarDate* ..... 333
- Identifier field ..... 286
- IF\_ABAP\_BEHAVIOR\_TESTDOUBLE ... 667
- IF\_CDS\_ACCESS\_CONTROL\_DOUBLE->  
DISABLE\_ACCESS\_CONTROL ..... 645
- IF\_CDS\_ACCESS\_CONTROL\_DOUBLE->  
>ENABLE\_ACCESS\_CONTROL ..... 644
- IF\_CDS\_TEST\_ENVIRONMENT ..... 659
- IF\_CDS\_TEST\_ENVIRONMENT->  
CLEAR\_DOUBLES ..... 640
- IF\_CDS\_TEST\_ENVIRONMENT->  
DESTROY ..... 643
- ina* ..... 230
- Inconsistency ..... 152
  - remove* ..... 153
  - technical* ..... 154
- Indicator ..... 287
- indicators null structure ..... 688
- InfoObject name ..... 322, 358
  - local* ..... 295
- Information Access (InA) ..... 358
  - protocol* ..... 358
- Information Access (InA) UI
  - service* ..... 218, 222, 228

- Infrastructure
  - analytics* ..... 303, 356
  - transactional* ..... 363
- Inheritance mechanism ..... 182
  - side effects* ..... 182
- Instance authorization ..... 161
- Integration test ..... 207, 638, 670
- Interface behavior definition ..... 361, 478
- Interface view ..... 282
- Intersect element ..... 80
- is initial ..... 695
- is not null ..... 650, 690
- is null ..... 690

## J

- Join ..... 47, 81
  - cardinality* ..... 82, 84
  - cross* ..... 82
  - inner* ..... 82, 123, 124, 169
  - left outer* ... 59, 81, 84, 85, 123, 650, 689
  - multiple data sources* ..... 86
  - performance aspects* ..... 81
  - right outer* ..... 81

## K

- Key ..... 35, 50, 60, 61, 72
  - alternative* ..... 138
  - definitions* ..... 79
  - representative* ..... 291
- Key field ..... 61
  - representative* ..... 314, 316
- Key performance indicator (KPI) ..... 304
- Key user ..... 304, 600
- Key user application ..... 592, 600, 614
- Key user extensibility ..... 600, 614

## L

- Label text ..... 62
- LANGUAGE\_DEPENDENT\_TEXT ..... 595
- Leading model ..... 277
- leaf ..... 547
- Leaf of a hierarchy ..... 516
  - node* ..... 517
- Leveled hierarchy ..... 516
- Levels ..... 546
- Lifecycle ..... 585, 588, 590
  - periods* ..... 591
  - stability contract CO* ..... 602
- LimitedDescendantCount ..... 547

- Limit statement ..... 684
- Localized ..... 214, 216, 269, 331
  - elements* ..... 472
- Local naming ..... 358
- Lock ..... 624
  - node level* ..... 397
- Lock object ..... 393
- Logical unit of work (LUW) ..... 447, 495

## M

- Machine learning ..... 245
- MANAGED ..... 618, 621, 622
- Manual test ..... 207
- Mapping ..... 620
- MAX ..... 88, 259
- Maximum ..... 258
- Measure ..... 311, 346
  - analytic* ..... 306
  - calculate* ..... 338
  - restricted* ..... 342
- Memory overflow ..... 89
- Metadata ..... 143, 252
- Metadata extension ..... 57, 131, 155, 500, 622
  - create* ..... 156
  - layer assignment* ..... 157
  - names* ..... 156
  - permitted annotations* ..... 158
- Meta information ..... 249
- MIN ..... 88, 259
- Minimum ..... 258
- M-N relationship ..... 114
- Model consistency ..... 322
- Modeling errors ..... 141
- Modeling pattern ..... 102, 267, 595
- Modification ..... 602
- MODIFY ENTITY ..... 663
- Multiple parents ..... 537

## N

- Naming collisions ..... 612
- Naming rules ..... 219
- Node
  - business object* ..... 270
  - distance from root* ..... 517
  - object* ..... 270
  - orphaned* ..... 533, 536
  - type* ..... 523
  - view* ..... 520
- node\_id* ..... 533

NodeProperty ..... 546  
 NONE ..... 259  
 Non-null preserving expression ..... 68  
 NOT\_RELEASED ..... 590  
 NOT\_TO\_BE\_RELEASED ..... 590  
 NOT\_TO\_BE\_RELEASED\_STABLE ..... 590  
 Null preserving expression ..... 691  
 Null value ..... 68, 84, 86, 101, 140, 146,  
 148, 151, 193, 636, 649, 650, 688, 690  
   *annotating* ..... 140  
   *handling* ..... 85  
 Numbering ..... 361, 372  
   *early* ..... 387  
   *external* ..... 387, 401  
   *internal* ..... 388  
   *late* ..... 388  
 Number range ..... 388, 390, 495  
   *object* ..... 391

**O**

Object ..... 270  
   *model* ..... 366  
   *node* ..... 270  
 Object type  
   *code* ..... 278  
   *DDLX* ..... 155  
   *STOB* ..... 61  
 OData ..... 17, 358, 364, 365, 377, 383, 447,  
 449, 473, 480, 484, 492, 493, 497–499  
   *ABAP code exits* ..... 676  
   *publication* ..... 225  
   *SADL-based* ..... 676  
   *UI* ..... 499  
   *UI definition* ..... 231  
   *V2* ..... 497  
   *V4* ..... 497  
   *version* ..... 224  
   *Web API* ..... 498  
 OData entity ..... 143  
   *property* ..... 220  
   *set* ..... 57, 62, 219  
   *type* ..... 143, 220  
 OData navigation property ..... 224, 226  
 OData service ..... 35, 143, 251, 499,  
 616, 622, 626, 730  
   *hierarchical* ..... 544  
   *metadata* ..... 559, 626  
   *search* ..... 570  
 OLAP processor ..... 357  
 On condition ..... 41, 108, 113, 118, 121,  
 122, 125, 127

Open dialog box ..... 298  
 Operation ..... 658  
   *interaction phase* ..... 493  
   *save phase* ..... 494  
 Optimistic concurrency control ..... 447  
 Optional element ..... 186  
 Orphaned ..... 533  
 Orphans ..... 536  
 OUTPUT\_EMAIL\_DATA\_  
   PROVIDER ..... 597  
 OUTPUT\_FORM\_DATA\_  
   PROVIDER ..... 597  
 OUTPUT\_PARAMETER\_DETERMINA-  
   TION\_DATA\_SOURCE ..... 597  
 Output node ..... 533, 538, 539

**P**

Parameter ..... 70, 91, 178, 332  
   *association definitions* ..... 94  
   *names* ..... 92  
   *variable* ..... 336  
 PARENT\_CHILD\_HIERARCHY\_  
   NODE\_PROVIDER ..... 596  
 PARENT\_ID ..... 533  
 Parent association ..... 529, 530  
 Parent-child hierarchy ..... 516, 519  
 Parent node ..... 270, 519  
   *multiple* ..... 537  
 Parent relation ..... 516  
 Path expression ..... 44, 45, 107, 108, 111,  
 118, 121, 123, 129, 173, 205  
   *CDS role* ..... 179  
   *parameters* ..... 94  
 Performance ..... 58, 68, 78, 84, 89, 101,  
 104, 110, 144  
   *aspects* ..... 183, 193  
   *optimization* ..... 109, 122  
 Performance aspects ..... 168, 180, 671  
   *testing* ..... 678  
 Period From..To ..... 541  
 Persisted field value ..... 675  
 Persistency model ..... 677  
 pfcg\_mapping ..... 207  
 Placeholder ..... 242  
 Point in time ..... 260, 287  
 Precheck ..... 445, 621  
 Prediction procedure ..... 245  
 Predictive analytics ..... 245  
 Prefix  
   C\_ ..... 283  
   P\_ ..... 284

Prefix (Cont.)  
   R\_ ..... 282  
 Preserving type ..... 64, 128  
   *statement* ..... 62  
 Principle of least privilege ..... 161  
 Privileged access ..... 170, 204, 205  
 Problems tab ..... 700, 702, 704  
 Programming model ..... 250  
 Projected associations ..... 117  
 Project Explorer ..... 36, 38, 56, 105, 222  
 Projection ..... 467  
   *field* ..... 89  
   *view* ..... 211, 212, 269  
 Projection behavior  
   *actions* ..... 486  
   *definitions* ..... 361, 484  
   *operations* ..... 480  
   *precheck* ..... 487  
 Projection object model  
   *define* ..... 467  
 Propagation logic ..... 131, 145, 155  
 Provider contract ..... 102, 194, 197, 215,  
 230, 471, 477, 484, 549  
   *analytical\_query* ..... 326  
 Provider implementation ..... 381  
 Proxy object ..... 632

**Q**

Qualifier ..... 287  
 Qualifier of a name ..... 285  
 quan ..... 96  
 Quantity field ..... 257, 287  
 Query  
   *analytic* ..... 303, 325  
   *display attribute* ..... 195  
   *layout* ..... 329, 330  
   *monitor* ..... 326  
   *settings* ..... 325  
   *variable* ..... 195  
 Quick Assist ..... 657  
 Quick info ..... 255  
 Quota rules for extension fields ..... 605

**R**

Readability ..... 155  
 Read access ..... 166, 250  
 READ ENTITY ..... 663  
 Redefinition ..... 200  
 Redundancy ..... 281, 293

Reference data model ..... 24  
   *entities* ..... 24  
   *implementation* ..... 26  
 Reference fields ..... 96, 258  
 Regression ..... 207  
   *issue* ..... 631, 638  
 Regular expression ..... 244  
 Release ..... 585  
 Release contract ..... 585  
 RELEASED ..... 590  
 RELEASED\_WITH\_FEATURE\_  
   TOGGLE ..... 590  
 Release state ..... 588, 590, 605  
   *deprecated* ..... 592  
 Remote API view ..... 283  
 Replacing conditions with ..... 188  
 Replacing root with statement ..... 183  
 Replacing with statement ..... 188  
 Report RUTDLSV2MIGRATION ..... 56  
 Representational State Transfer  
   (Rest) ..... 362, 365, 450  
 Representation term ..... 286  
 Representative key field ..... 266, 269  
 Responsive design ..... 500  
 Restricted reuse view ..... 282  
 ROLLBACK WORK ..... 447  
 Root node ..... 112, 114, 270, 279, 516  
 Root view ..... 112  
 Runtime behavior ..... 291  
 Runtime orchestration ..... 361, 397, 398,  
 408, 492

**S**

SAP Analysis for Microsoft Office ..... 328  
 SAP API Deprecation Policy ..... 590  
 SAP BTP, ABAP environment ..... 600, 613  
 SAP Business Accelerator Hub ..... 295  
 SAP Business Technology Platform  
   (SAP BTP) ..... 600  
 SAP CDS View Deprecation Policy ..... 590  
 SAP client ..... 237  
 SAP Event Mesh ..... 362, 510, 512  
 SAP Fiori ..... 250, 364, 449, 463, 484,  
 498, 730  
   *application* ..... 17, 499, 500  
   *architecture* ..... 250  
 SAP Fiori apps reference library ..... 298  
 SAP Fiori elements ..... 231, 252, 362, 376,  
 436, 450, 458, 499, 503, 510, 566, 628  
 SAP Gateway ..... 253  
 SAP Gateway Service Builder ..... 212, 735

SAP GUI ..... 694  
 SAP HANA ..... 277, 290, 364  
   *conversion functions* ..... 63  
   *database* ..... 17  
   *execution plan* ..... 672  
   *native functions* ..... 235  
   *optimizer* ..... 68, 109, 675  
   *SQLScript* ..... 57  
   *table function* ..... 235, 239  
 SAP HANA Studio ..... 241  
 SAP List Viewer (ALV) ..... 533  
 SAP object node type ..... 146, 275, 279  
 SAP object type ..... 275, 277, 513  
   *root node* ..... 279  
 SAP S/4HANA ..... 17, 361, 363, 364  
   *analytics* ..... 303  
   *architecture* ..... 250, 253  
   *programming model* ..... 249  
   *virtual data model* ..... 275, 363  
 SAPUI5 ..... 488  
 Search ..... 549  
   *check* ..... 574  
   *field* ..... 571, 579  
   *request* ..... 573  
   *results* ..... 573  
   *scope* ..... 575  
 SEARCHABLE\_ENTITY ..... 596  
 Select distinct statement ..... 72  
 Select from statement ..... 34, 59  
 Select statement ..... 43, 58, 62, 107, 166  
   *associations* ..... 107  
   *change* ..... 44  
   *optimize* ..... 675  
   *SAP HANA database* ..... 678  
 Semantic node ..... 533, 539  
 Service Adaptation Definition  
   Language (SADL) ... 35, 54, 57, 143, 205,  
   212, 365, 473, 549, 557, 558, 571, 682  
 Service binding ..... 211, 221, 497, 628  
   *name* ..... 223  
   *type* ..... 223  
 Service definition ..... 211, 216, 626  
   *names* ..... 218  
   *specialized CDS entities* ..... 219  
 Service endpoint ..... 225  
 Service infrastructure ..... 253  
 Service provider contract ..... 230  
 Session variable ..... 69, 70, 237, 238, 243  
   *CDS\_CLIENT* ..... 289  
 Setup method ..... 640, 645  
 SIBLINGS ORDER ..... 530, 531  
 Side effect ..... 361, 427, 437, 458, 491,  
   494, 619, 620  
 Simple Object Access Protocol  
   (SOAP) ..... 492  
 Simple type ..... 54, 55, 58, 66, 253, 463  
 SITUATION\_ANCHOR ..... 597  
 SITUATION\_DATACONTEXT ..... 597  
 SITUATION\_TRIGGER ..... 597  
 Smart control ..... 252  
 Smart template ..... 252  
 SOURCE ..... 530  
 SQL\_DATA\_SOURCE ..... 595  
 SQL aggregation function ..... 87  
   *performance aspects* ..... 89  
 SQL Console ..... 43, 243  
 SQLScript ..... 20, 235, 245  
 SQL Web API service ..... 230  
 Stability contract ..... 483, 585, 586, 588  
   *C0* ..... 586, 588, 599, 602, 604–607, 615  
   *C1* ..... 281, 290, 483, 586, 588, 608  
   *C2* ..... 586, 588  
   *C3* ..... 586  
   *C4* ..... 586  
 Stable data source ..... 608  
 Stable extension ..... 608  
 Standard aggregation ..... 258  
   *behavior* ..... 311  
   *types* ..... 259  
 Standard query ..... 308  
 Standard selection ..... 71  
 standard traversal ..... 531  
 Star schema ..... 322  
 Start authorization ..... 161  
 Start condition ..... 530  
 START WHERE ..... 530, 531  
 statement inheriting conditions  
   from super ..... 201  
 Static complexity ..... 672  
 Static field control ..... 386, 479  
 Static operation control ..... 386  
 STRING\_AGG ..... 244  
 Structured query language (SQL) ..... 23,  
   53, 235, 473  
   *CREATE statement* ..... 36  
   *dependency graph* ..... 46  
   *dependency tree* ..... 46  
   *functions* ..... 442  
   *select request* ..... 251  
 Structure include ..... 603, 604, 606–608,  
   610, 611, 614  
 Subnotation ..... 136  
 Successor ..... 592

Suffix for extension fields ..... 605  
 SUM ..... 88, 259  
 Summation ..... 258  
 Supported capability ..... 280, 585, 594  
 sy-datlo ..... 70  
 sy-datum ..... 70, 94  
 sy-langu ..... 70  
 sy-mandt ..... 70  
 Syntax check ..... 102  
 System field ..... 144, 336  
 System load ..... 291  
 System time-dependency ..... 272  
 System times ..... 260  
 sy-uname ..... 70  
 sy-zonlo ..... 70

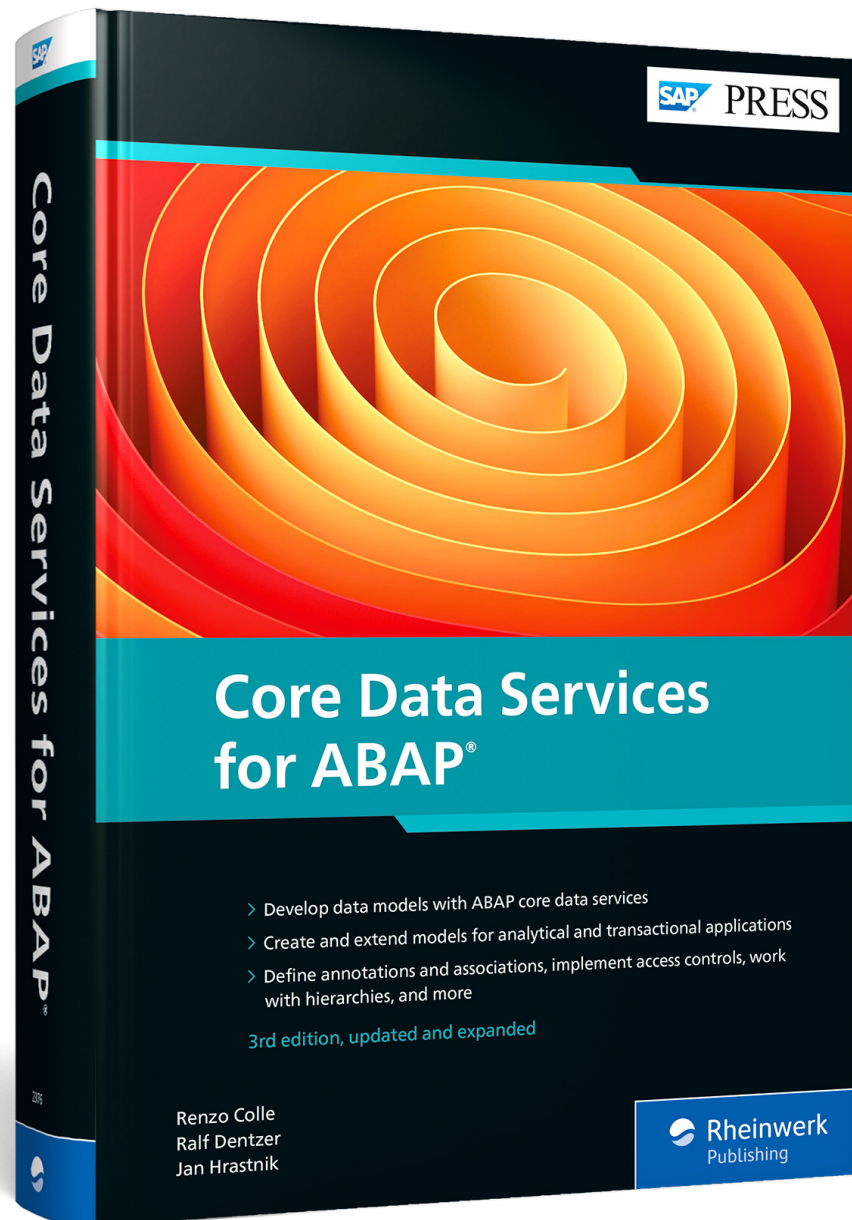
**T**

Table buffer ..... 58  
 Table definition ..... 30  
 Table function ..... 63  
 Template CDS model ..... 33  
 Temporal data ..... 272  
 Test ABAP code ..... 650  
 Test automation ..... 631  
 Test classes ..... 653  
   *adaptation* ..... 657  
   *Quick Assist* ..... 657  
 Test double ..... 659  
   *behavior* ..... 667, 668  
   *event* ..... 662  
 Test double framework ..... 632, 636, 651  
   *decoupling option* ..... 637, 638  
 Test environment ..... 326  
   *analytic views* ..... 307  
 Text ..... 261  
   *association* ..... 269, 316  
   *denormalization* ..... 214  
   *relations* ..... 267  
   *view* ..... 269, 315, 316, 320  
 Time-dependent data ..... 272  
 TopLevels ..... 546  
 Trace Configurations tab ..... 711  
 Trace Results tab ..... 711, 712  
 Transaction  
   */IWFND/GW\_CLIENT* ..... 545  
   */IWFND/MAINT\_SERVICE* ..... 225  
   */IWFND/V4\_ADMIN* ..... 226  
   *CDS\_REPAIR* ..... 708, 711  
   *PFCG* ..... 164  
   *RSRT* ..... 326, 328  
   *RSRTS\_ODP* ..... 524  
 Transaction (Cont.)  
   *RSRTS\_ODP\_DIS* ..... 307, 318, 321, 524  
   *SA38* ..... 308  
   *SACMSEL* ..... 207  
   *SE11* ..... 28  
   *SE38* ..... 308  
   *SEGW* ..... 212  
   *ST05* ..... 680, 684  
 TRANSACTIONAL\_INTERFACE ..... 102,  
   103, 197, 212, 597  
 TRANSACTIONAL\_PROVIDER ..... 597  
 TRANSACTIONAL\_QUERY ..... 103, 104,  
   212, 597  
 Transactional application ..... 361  
 Transactional consistency ..... 362  
 Transactional object model ..... 361, 366,  
   445, 467  
   *calculated fields* ..... 444  
   *data determinations and*  
     *validations* ..... 427  
   *locks* ..... 393  
   *restrict* ..... 467  
 Transactional projection  
   model ..... 361, 445  
 Transactional view layer ..... 367  
 Transient projection view ... 328, 341, 350  
 Transient view ..... 215, 228  
 Transport object ..... 33  
 Tree table ..... 544  
 Trigger condition ..... 428, 432, 436, 459  
 Troubleshooting ..... 671  
   *activation issues* ..... 706  
   *incorrect annotation* ..... 700  
 Type category ..... 278  
 Typed literal ..... 64  
 Type extension ..... 603, 611, 614  
 Type-preserving cast ..... 65

**U**

UI annotation ..... 252, 499  
 UI orchestration ..... 452  
 UI service ..... 218, 222  
 Union All logic ..... 78  
 Union statement ..... 73, 78  
 Union views ..... 73  
   *association definitions* ..... 77  
   *two-layer construction* ..... 76  
 Unit conversion ..... 99, 100  
 Unit field ..... 96, 257  
 Unit test ..... 662, 669

- Universally unique identifier  
(UUID) ..... 287
- Upgrade issues ..... 602
- using client ..... 71
- utcl\_current ..... 104
- ## V
- Validation ..... 383, 387, 428, 433, 452, 454, 618
- VALUE\_HELP\_PROVIDER ..... 596
- Value help ..... 549, 575, 596
- associated view ..... 555
  - expose ..... 558
  - integrate ..... 553
  - modeling ..... 549, 550
  - nested ..... 552, 562
  - OData ..... 559
  - service binding type ..... 558
  - UI applications ..... 566
  - usage ..... 559
- ValueListMapping ..... 561
- ValueListParameterDisplay-  
Only ..... 562, 563
- ValueListParameterInOut ... 562, 563, 566
- ValueListParameterOut ..... 564
- Value view ..... 263
- Variable ..... 332
- analytic ..... 332
  - derivation ..... 337
  - values ..... 335
- VDM view
- find ..... 295
  - private ..... 284
  - reuse ..... 277
- View
- ABAP cross trace ..... 711
  - active annotations ..... 50, 146, 152
  - analytic ..... 303, 305
  - annotation ..... 288
  - CDS navigator ..... 49
  - data preview ..... 685, 704
- View (Cont.)
- I\_SalesOrder ..... 292
  - migration ..... 56
  - outline ..... 48
  - project explorer ..... 36, 217, 222, 654, 656
  - properties ..... 39
  - relation explorer ..... 49
  - reuse ..... 123
  - search ..... 48
  - SQL Console ..... 684, 686
  - static complexity ..... 122, 123, 179
  - structure ..... 292
- View Browser ..... 295, 297
- View entity ..... 307, 327
- virtual ..... 342
- Virtual data model (VDM) ..... 17, 26, 218, 275, 305, 363, 578
- deprecation ..... 592
  - layers ..... 280
  - naming ..... 285
  - principles ..... 277
  - structure ..... 280
  - types ..... 284
- Virtual element ..... 474
- Virtual field ..... 97
- Visibility ..... 602
- void ..... 188
- ## W
- Web API
- OData ..... 498
  - service ..... 227
- Where statement ..... 60
- Where-used list ..... 48, 301
- With optional elements ..... 186
- with parameters ..... 91
  - with privileged access ..... 204, 205
  - with user element ..... 191
- Wrapper view ..... 243
- Write access ..... 250



**Renzo Colle** is currently responsible for the end-to-end programming model of SAP S/4HANA in the central architecture group. He studied business mathematics at the Karlsruhe Institute of Technology (<https://www.kit.edu/english>) and has worked at SAP for more than 25 years in a wide variety of areas and roles. He started his career at SAP as a developer in strategic customer development. In SAP Business ByDesign, he was responsible for logistics and lead architect of the SAP ByDesign platform for cloud applications. As the inventor of the Business Object Processing Framework (BOPF) and lead architect of the ABAP RESTful application programming model, he has worked on model-driven software development and transactional applications for more than 20 years. Further information about Renzo can be found at <https://de.linkedin.com/in/renzo-colle-30804ba1/de>.



**Ralf Dentzer** has been working for several years in the central architecture group of the SAP S/4HANA suite with a focus on the use of core data services (CDS) in SAP S/4HANA. He joined SAP more than 25 years ago. He developed HR applications for SAP R/3, SAP ERP, and SAP Business ByDesign. After that, his tasks shifted to questions of overall architecture for new solutions. Ralf studied mathematics and received his doctorate from the University of Heidelberg. He is married and has two adult sons.



**Jan Hrastnik** is a member of the SAP S/4HANA cross architecture team, where he focuses on the virtual data model (VDM) and the use of CDS in ABAP applications. He has worked in various SAP development areas for more than 20 years. At the beginning of his career, he supported numerous customer projects in the automotive industry. Subsequently, he worked in the supply chain management development of SAP Business ByDesign. Jan's work initially focused on developing the master data required for the production processes before he took on overarching expert tasks for central architecture topics. He then worked on the SAP SuccessFactors Employee Central solution and native SAP HANA application development.

Colle, Dentzer, Hrastnik

## Core Data Services for ABAP

754 pages | 10/2023 | \$89.95 | ISBN 978-1-4932-2376-3

 [www.sap-press.com/5642](http://www.sap-press.com/5642)

*We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.*